# Contents

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Chapter 1</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>About the EWL C++ Library Reference Manual</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 2</strong></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>The EWL C++ Library Overview</td>
<td>57</td>
</tr>
<tr>
<td>2.2</td>
<td>Definitions</td>
<td></td>
</tr>
<tr>
<td>2.2.1</td>
<td>Arbitrary-Positional Stream</td>
<td>58</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Character</td>
<td>58</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Character Sequences</td>
<td>58</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Comparison Function</td>
<td>58</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Component</td>
<td>59</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Default Behavior</td>
<td>59</td>
</tr>
<tr>
<td>2.2.7</td>
<td>Handler Function</td>
<td>59</td>
</tr>
<tr>
<td>2.2.8</td>
<td>Iostream Class Templates</td>
<td>59</td>
</tr>
<tr>
<td>2.2.9</td>
<td>Modifier Function</td>
<td>59</td>
</tr>
<tr>
<td>2.2.10</td>
<td>Object State</td>
<td>59</td>
</tr>
<tr>
<td>2.2.11</td>
<td>Narrow-oriented Iostream Classes</td>
<td>60</td>
</tr>
<tr>
<td>2.2.12</td>
<td>NTCTS</td>
<td>60</td>
</tr>
<tr>
<td>2.2.13</td>
<td>Observer Function</td>
<td>60</td>
</tr>
<tr>
<td>2.2.14</td>
<td>Replacement Function</td>
<td>60</td>
</tr>
<tr>
<td>2.2.15</td>
<td>Required Behavior</td>
<td>60</td>
</tr>
<tr>
<td>2.2.16</td>
<td>Repositional Stream</td>
<td>60</td>
</tr>
<tr>
<td>2.2.17</td>
<td>Reserved Function</td>
<td>61</td>
</tr>
<tr>
<td>2.2.18</td>
<td>Traits</td>
<td>61</td>
</tr>
<tr>
<td>2.2.19</td>
<td>Wide-oriented IOSTREAM Classes</td>
<td>61</td>
</tr>
<tr>
<td>2.3</td>
<td>Additional Definitions</td>
<td>61</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Multi-Thread Safety</td>
<td>61</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>2.3.1.1</td>
<td>EWL C++ Thread Safety Policy</td>
<td>62</td>
</tr>
<tr>
<td>2.4</td>
<td>Methods of Descriptions</td>
<td>63</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Structure of each sub-clause</td>
<td>63</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Other Conventions</td>
<td>63</td>
</tr>
<tr>
<td>2.4.2.1</td>
<td>Character sequences</td>
<td>63</td>
</tr>
<tr>
<td>2.4.2.2</td>
<td>Byte strings</td>
<td>63</td>
</tr>
<tr>
<td>2.4.2.3</td>
<td>Multibyte strings</td>
<td>64</td>
</tr>
<tr>
<td>2.4.2.4</td>
<td>Wide-character sequences</td>
<td>64</td>
</tr>
<tr>
<td>2.4.2.5</td>
<td>Functions within classes</td>
<td>64</td>
</tr>
<tr>
<td>2.4.2.6</td>
<td>Private members</td>
<td>64</td>
</tr>
<tr>
<td>2.5</td>
<td>Library-wide Requirements</td>
<td>65</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Library contents and organization</td>
<td>65</td>
</tr>
<tr>
<td>2.5.1.1</td>
<td>Library Contents</td>
<td>65</td>
</tr>
<tr>
<td>2.5.1.2</td>
<td>Headers</td>
<td>65</td>
</tr>
<tr>
<td>2.5.1.3</td>
<td>Freestanding Implementations</td>
<td>66</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Using the library</td>
<td>67</td>
</tr>
<tr>
<td>2.5.2.1</td>
<td>Headers</td>
<td>67</td>
</tr>
<tr>
<td>2.5.2.2</td>
<td>Linkage</td>
<td>67</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Constraints on programs</td>
<td>67</td>
</tr>
<tr>
<td>2.5.3.1</td>
<td>Reserved Names</td>
<td>67</td>
</tr>
<tr>
<td>2.5.3.2</td>
<td>External Linkage</td>
<td>68</td>
</tr>
<tr>
<td>2.5.3.3</td>
<td>Headers</td>
<td>68</td>
</tr>
<tr>
<td>2.5.3.4</td>
<td>Derived classes</td>
<td>68</td>
</tr>
<tr>
<td>2.5.3.5</td>
<td>Replacement Functions</td>
<td>68</td>
</tr>
<tr>
<td>2.5.3.6</td>
<td>Handler functions</td>
<td>69</td>
</tr>
<tr>
<td>2.5.3.7</td>
<td>Other functions</td>
<td>69</td>
</tr>
<tr>
<td>2.5.3.8</td>
<td>Function arguments</td>
<td>69</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Conforming Implementations</td>
<td>70</td>
</tr>
</tbody>
</table>
# Language Support Library

## 3.1 Types

- Section 3.1 Types (71)

## 3.2 Implementation properties

- Section 3.2 Implementation properties (72)
  - 3.2.1 Numeric limits (72)
  - 3.2.2 `is_specialized` (72)
  - 3.2.3 `min` (72)
  - 3.2.4 `max` (73)
  - 3.2.5 `digits` (73)
  - 3.2.6 `is_signed` (73)
  - 3.2.7 `is_integer` (73)
  - 3.2.8 `is_exact` (74)
  - 3.2.9 `radix` (74)
  - 3.2.10 `epsilon` (74)
  - 3.2.11 `round_error` (74)
  - 3.2.12 `min_exponent` (75)
  - 3.2.13 `min_exponent10` (75)
  - 3.2.14 `max_exponent` (75)
  - 3.2.15 `max_exponent10` (75)
  - 3.2.16 `has_infinity` (76)
  - 3.2.17 `has_quiet_NaN` (76)
  - 3.2.18 `has_signaling_NaN` (76)
  - 3.2.19 `has_denorm` (76)
  - 3.2.20 `has_denorm_loss` (77)
  - 3.2.21 `infinity` (77)
  - 3.2.22 `quiet_NaN` (77)
  - 3.2.23 `signaling_NaN` (77)
  - 3.2.24 `denorm_min` (78)
  - 3.2.25 `is_iec559` (78)
  - 3.2.26 `is_bounded` (78)
3.2.27 is_modulo..........................................................................................................................................................78
3.2.28 traps..................................................................................................................................................................79
3.2.29 tinyness_before................................................................................................................................................79
3.2.30 round_style....................................................................................................................................................79
3.2.31 Type float_round_style................................................................................................................................79
3.2.32 Type float_denorm_style................................................................................................................................80
3.2.33 numeric_limits specializations........................................................................................................................80
3.3 Start and termination....................................................................................................................................................81
3.3.1 abort...................................................................................................................................................................81
3.3.2 atexit....................................................................................................................................................................82
3.3.3 exit....................................................................................................................................................................82
3.4 Dynamic Memory Management..................................................................................................................................83
3.4.1 Storage Allocation and Deallocation............................................................................................................83
3.4.2 Single Object Forms........................................................................................................................................83
  3.4.2.1 operator new................................................................................................................................................83
  3.4.2.2 operator delete...........................................................................................................................................84
3.4.3 Array Forms.....................................................................................................................................................84
  3.4.3.1 operator new[]..............................................................................................................................................84
  3.4.3.2 operator delete[]..................................................................................................................................85
3.4.4 Placement Forms..............................................................................................................................................85
  3.4.4.1 Placement operator new..........................................................................................................................85
  3.4.4.2 Placement operator delete.......................................................................................................................85
3.4.5 Storage Allocation Errors.....................................................................................................................................85
  3.4.5.1 Class Bad_alloc...........................................................................................................................................86
  3.4.5.2 Constructor...............................................................................................................................................86
  3.4.5.3 Assignment Operator.............................................................................................................................86
  3.4.5.4 destructor................................................................................................................................................86
  3.4.5.5 what........................................................................................................................................................86
  3.4.5.6 type new_handler......................................................................................................................................87
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.5.7</td>
<td>set_new_handler.................................................................</td>
<td>87</td>
</tr>
<tr>
<td>3.5</td>
<td>Type identification..............................................................</td>
<td>87</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Class type_info...........................................................................</td>
<td>88</td>
</tr>
<tr>
<td>3.5.1.1</td>
<td>Constructors...............................................................................</td>
<td>88</td>
</tr>
<tr>
<td>3.5.1.2</td>
<td>Assignment Operator....................................................................</td>
<td>88</td>
</tr>
<tr>
<td>3.5.1.3</td>
<td>operator==..............................................................................</td>
<td>88</td>
</tr>
<tr>
<td>3.5.1.4</td>
<td>operator!=...............................................................................</td>
<td>88</td>
</tr>
<tr>
<td>3.5.1.5</td>
<td>before.....................................................................................</td>
<td>89</td>
</tr>
<tr>
<td>3.5.1.6</td>
<td>name......................................................................................</td>
<td>89</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Class bad_cast...........................................................................</td>
<td>89</td>
</tr>
<tr>
<td>3.5.2.1</td>
<td>Constructors...............................................................................</td>
<td>89</td>
</tr>
<tr>
<td>3.5.2.2</td>
<td>Assignment Operator....................................................................</td>
<td>89</td>
</tr>
<tr>
<td>3.5.2.3</td>
<td>what......................................................................................</td>
<td>90</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Class bad_typeid.........................................................................</td>
<td>90</td>
</tr>
<tr>
<td>3.5.3.1</td>
<td>Constructors...............................................................................</td>
<td>90</td>
</tr>
<tr>
<td>3.5.3.2</td>
<td>Assignment Operator....................................................................</td>
<td>90</td>
</tr>
<tr>
<td>3.5.3.3</td>
<td>what......................................................................................</td>
<td>90</td>
</tr>
<tr>
<td>3.6</td>
<td>Exception Handling...................................................................</td>
<td>91</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Class exception........................................................................</td>
<td>91</td>
</tr>
<tr>
<td>3.6.1.1</td>
<td>Constructors...............................................................................</td>
<td>91</td>
</tr>
<tr>
<td>3.6.1.2</td>
<td>Assignment Operator....................................................................</td>
<td>91</td>
</tr>
<tr>
<td>3.6.1.3</td>
<td>destructor...............................................................................</td>
<td>91</td>
</tr>
<tr>
<td>3.6.1.4</td>
<td>what......................................................................................</td>
<td>92</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Violating Exception Specifications...........................................</td>
<td>92</td>
</tr>
<tr>
<td>3.6.2.1</td>
<td>Class bad_exception....................................................................</td>
<td>92</td>
</tr>
<tr>
<td>3.6.2.1.1</td>
<td>Constructors...............................................................................</td>
<td>92</td>
</tr>
<tr>
<td>3.6.2.1.2</td>
<td>Assignment Operator....................................................................</td>
<td>92</td>
</tr>
<tr>
<td>3.6.2.1.3</td>
<td>what......................................................................................</td>
<td>92</td>
</tr>
<tr>
<td>3.6.2.1.4</td>
<td>type unexpected_handler.........................................................</td>
<td>93</td>
</tr>
</tbody>
</table>
Section number | Title | Page
--- | --- | ---
3.6.2.1.5 | set_unexpected | 93
3.6.2.1.6 | unexpected | 93
3.7 | Abnormal Termination | 93
3.7.1 | type terminate_handler | 93
3.7.2 | set_terminate | 94
3.7.3 | terminate | 94
3.7.4 | uncaught_exception | 94
3.8 | Other Runtime Support | 95

Chapter 4
Diagnostics Library

4.1 | Exception Classes | 97
4.1.1 | Class logic_error | 97
4.1.2 | Class domain_error | 98
4.1.3 | Class invalid_argument | 98
4.1.4 | Class length_error | 98
4.1.5 | Class out_of_range | 99
4.1.6 | Class runtime_error | 99
4.1.7 | Class range_error | 99
4.1.8 | Class overflow_error | 100
4.1.9 | Class underflow_error | 100
4.2 | Assertions | 100
4.3 | Error Numbers | 101

Chapter 5
General Utilities Libraries

5.1 | Requirements | 103
5.1.1 | Equality Comparisons | 103
5.1.2 | Less Than Comparison | 104
5.1.3 | Copy Construction | 104
5.1.4 | Default Construction | 104
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.5</td>
<td>Allocator Requirements</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Utility Components</td>
<td>105</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Operators</td>
<td>106</td>
</tr>
<tr>
<td>5.2.1.1</td>
<td>operator!=</td>
<td>106</td>
</tr>
<tr>
<td>5.2.1.2</td>
<td>operator&gt;</td>
<td>106</td>
</tr>
<tr>
<td>5.2.1.3</td>
<td>operator&lt;=</td>
<td>106</td>
</tr>
<tr>
<td>5.2.1.4</td>
<td>operator&gt;=</td>
<td>106</td>
</tr>
<tr>
<td>5.3</td>
<td>Pairs</td>
<td>107</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Constructors</td>
<td>107</td>
</tr>
<tr>
<td>5.3.2</td>
<td>operator ==</td>
<td>107</td>
</tr>
<tr>
<td>5.3.3</td>
<td>operator &lt;</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>make_pair</td>
<td>108</td>
</tr>
<tr>
<td>5.4</td>
<td>Function objects</td>
<td>108</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Arithmetic operations</td>
<td>108</td>
</tr>
<tr>
<td>5.4.1.1</td>
<td>plus</td>
<td>109</td>
</tr>
<tr>
<td>5.4.1.2</td>
<td>minus</td>
<td>109</td>
</tr>
<tr>
<td>5.4.1.3</td>
<td>multiplies</td>
<td>109</td>
</tr>
<tr>
<td>5.4.1.4</td>
<td>divides</td>
<td>110</td>
</tr>
<tr>
<td>5.4.1.5</td>
<td>modulus</td>
<td>110</td>
</tr>
<tr>
<td>5.4.1.6</td>
<td>negate</td>
<td>110</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Comparisons</td>
<td>110</td>
</tr>
<tr>
<td>5.4.2.1</td>
<td>equal_to</td>
<td>111</td>
</tr>
<tr>
<td>5.4.2.2</td>
<td>not_equal_to</td>
<td>111</td>
</tr>
<tr>
<td>5.4.2.3</td>
<td>greater</td>
<td>111</td>
</tr>
<tr>
<td>5.4.2.4</td>
<td>less</td>
<td>112</td>
</tr>
<tr>
<td>5.4.2.5</td>
<td>greater_equal</td>
<td>112</td>
</tr>
<tr>
<td>5.4.2.6</td>
<td>less_equal</td>
<td>112</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Logical operations</td>
<td>112</td>
</tr>
<tr>
<td>5.4.3.1</td>
<td>logical_and</td>
<td>113</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>5.4.3.2</td>
<td>logical_or</td>
<td>113</td>
</tr>
<tr>
<td>5.4.3.3</td>
<td>logical_not</td>
<td>113</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Negators</td>
<td>114</td>
</tr>
<tr>
<td>5.4.4.1</td>
<td>Unary_negate</td>
<td>114</td>
</tr>
<tr>
<td>5.4.4.2</td>
<td>binary_negate</td>
<td>114</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Binders</td>
<td>115</td>
</tr>
<tr>
<td>5.4.5.1</td>
<td>Template class binder1st</td>
<td>115</td>
</tr>
<tr>
<td>5.4.5.2</td>
<td>bind1st</td>
<td>115</td>
</tr>
<tr>
<td>5.4.5.3</td>
<td>Template class binder2nd</td>
<td>115</td>
</tr>
<tr>
<td>5.4.5.4</td>
<td>bind2nd</td>
<td>116</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Adaptors for Pointers to Functions</td>
<td>116</td>
</tr>
<tr>
<td>5.4.6.1</td>
<td>pointer_to_unary_function</td>
<td>116</td>
</tr>
<tr>
<td>5.4.6.2</td>
<td>class pointer_to_binary_function</td>
<td>116</td>
</tr>
<tr>
<td>5.4.6.3</td>
<td>pointer_to_binary_function</td>
<td>116</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Adaptors for Pointers to Members</td>
<td>117</td>
</tr>
<tr>
<td>5.4.7.1</td>
<td>mem_fun_t</td>
<td>117</td>
</tr>
<tr>
<td>5.4.7.2</td>
<td>mem_fun1_t</td>
<td>117</td>
</tr>
<tr>
<td>5.4.7.3</td>
<td>mem_fun</td>
<td>118</td>
</tr>
<tr>
<td>5.4.7.4</td>
<td>mem_fun_ref_t</td>
<td>118</td>
</tr>
<tr>
<td>5.4.7.5</td>
<td>mem_fun1_ref_t</td>
<td>118</td>
</tr>
<tr>
<td>5.4.7.6</td>
<td>mem_fun_ref</td>
<td>118</td>
</tr>
<tr>
<td>5.4.7.7</td>
<td>const_mem_fun_t</td>
<td>119</td>
</tr>
<tr>
<td>5.4.7.8</td>
<td>const_mem_fun1_t</td>
<td>119</td>
</tr>
<tr>
<td>5.4.7.9</td>
<td>const_mem_fun_ref_t</td>
<td>120</td>
</tr>
<tr>
<td>5.4.7.10</td>
<td>const_mem_fun1_ref_t</td>
<td>120</td>
</tr>
<tr>
<td>5.5</td>
<td>Memory</td>
<td>120</td>
</tr>
<tr>
<td>5.5.1</td>
<td>allocator members</td>
<td>120</td>
</tr>
<tr>
<td>5.5.1.1</td>
<td>address</td>
<td>121</td>
</tr>
<tr>
<td>5.5.1.2</td>
<td>allocate</td>
<td>121</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.5.1.3</td>
<td>deallocate..................................</td>
<td>121</td>
</tr>
<tr>
<td>5.5.1.4</td>
<td>max_size....................................</td>
<td>121</td>
</tr>
<tr>
<td>5.5.1.5</td>
<td>construct....................................</td>
<td>122</td>
</tr>
<tr>
<td>5.5.1.6</td>
<td>destroy......................................</td>
<td>122</td>
</tr>
<tr>
<td>5.5.2</td>
<td>allocator globals..........................</td>
<td>122</td>
</tr>
<tr>
<td>5.5.2.1</td>
<td>operator==...................................</td>
<td>122</td>
</tr>
<tr>
<td>5.5.2.2</td>
<td>operator!=...................................</td>
<td>123</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Raw storage iterator.....................</td>
<td>123</td>
</tr>
<tr>
<td>5.5.3.1</td>
<td>Constructors..................................</td>
<td>123</td>
</tr>
<tr>
<td>5.5.3.2</td>
<td>operator *....................................</td>
<td>124</td>
</tr>
<tr>
<td>5.5.3.3</td>
<td>operator=.....................................</td>
<td>124</td>
</tr>
<tr>
<td>5.5.3.4</td>
<td>operator++...................................</td>
<td>124</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Temporary buffers..........................</td>
<td>125</td>
</tr>
<tr>
<td>5.5.4.1</td>
<td>get_temporary_buffer....................</td>
<td>125</td>
</tr>
<tr>
<td>5.5.4.2</td>
<td>return_temporary_buffer................</td>
<td>125</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Specialized Algorithms...................</td>
<td>125</td>
</tr>
<tr>
<td>5.5.5.1</td>
<td>uninitialized_copy......................</td>
<td>125</td>
</tr>
<tr>
<td>5.5.5.2</td>
<td>uninitialized_fill......................</td>
<td>126</td>
</tr>
<tr>
<td>5.5.5.3</td>
<td>uninitialized_fill_n....................</td>
<td>126</td>
</tr>
<tr>
<td>5.6</td>
<td>Template Class auto_ptr..................</td>
<td>126</td>
</tr>
<tr>
<td>5.6.1</td>
<td>auto_ptr constructors...................</td>
<td>129</td>
</tr>
<tr>
<td>5.6.2</td>
<td>operator =...................................</td>
<td>129</td>
</tr>
<tr>
<td>5.6.3</td>
<td>destructor...................................</td>
<td>130</td>
</tr>
<tr>
<td>5.6.4</td>
<td>auto_ptr Members..........................</td>
<td>130</td>
</tr>
<tr>
<td>5.6.5</td>
<td>operator*....................................</td>
<td>130</td>
</tr>
<tr>
<td>5.6.6</td>
<td>operator-&gt;(..................................</td>
<td>130</td>
</tr>
<tr>
<td>5.6.7</td>
<td>get...........................................</td>
<td>130</td>
</tr>
<tr>
<td>5.6.8</td>
<td>release......................................</td>
<td>131</td>
</tr>
<tr>
<td>5.6.9</td>
<td>reset.........................................</td>
<td>131</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>5.6.10</td>
<td>auto_ptr conversions</td>
<td>131</td>
</tr>
<tr>
<td>5.6.10.1</td>
<td>Conversion Constructor</td>
<td>131</td>
</tr>
<tr>
<td>5.6.10.2</td>
<td>operator auto_ptr_ref</td>
<td>132</td>
</tr>
<tr>
<td>5.6.11</td>
<td>operator auto_ptr</td>
<td>132</td>
</tr>
<tr>
<td>5.7</td>
<td>C Library</td>
<td>132</td>
</tr>
<tr>
<td>5.8</td>
<td>Date and Time</td>
<td>132</td>
</tr>
</tbody>
</table>

Chapter 6
Strings Library

6.1 Character traits | 135 |
6.1.1 Character Trait Definitions | 135 |
6.1.1.1 character | 136 |
6.1.1.2 character container type | 136 |
6.1.1.3 traits | 136 |
6.1.1.4 NTCTS | 136 |
6.1.2 Character Trait Requirements | 136 |
6.1.2.1 assign | 136 |
6.1.2.2 eq | 137 |
6.1.2.3 lt | 137 |
6.1.2.4 compare | 137 |
6.1.2.5 length | 137 |
6.1.2.6 find | 137 |
6.1.2.7 move | 138 |
6.1.2.8 copy | 138 |
6.1.2.9 not_eof | 138 |
6.1.2.10 to_char_type | 138 |
6.1.2.11 to_int_type | 138 |
6.1.2.12 eq_int_type | 139 |
6.1.2.13 get_state | 139 |
6.1.2.14 eof | 139 |
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.3</td>
<td>Character Trait Type Definitions.</td>
<td>139</td>
</tr>
<tr>
<td>6.1.4</td>
<td>struct char_traits&lt;T&gt;</td>
<td>140</td>
</tr>
<tr>
<td>6.2</td>
<td>String Classes</td>
<td>140</td>
</tr>
<tr>
<td>6.3</td>
<td>Class basic_string</td>
<td>140</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Constructors and Assignments</td>
<td>141</td>
</tr>
<tr>
<td>6.3.1.1</td>
<td>Constructors</td>
<td>141</td>
</tr>
<tr>
<td>6.3.1.2</td>
<td>Destructor</td>
<td>143</td>
</tr>
<tr>
<td>6.3.1.3</td>
<td>Assignment Operator</td>
<td>143</td>
</tr>
<tr>
<td>6.3.1.4</td>
<td>Assignment &amp; Addition Operator basic_string</td>
<td>143</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Iterator Support</td>
<td>144</td>
</tr>
<tr>
<td>6.3.2.1</td>
<td>begin</td>
<td>144</td>
</tr>
<tr>
<td>6.3.2.2</td>
<td>end</td>
<td>144</td>
</tr>
<tr>
<td>6.3.2.3</td>
<td>rbegin</td>
<td>144</td>
</tr>
<tr>
<td>6.3.2.4</td>
<td>rend</td>
<td>144</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Capacity</td>
<td>145</td>
</tr>
<tr>
<td>6.3.3.1</td>
<td>size</td>
<td>145</td>
</tr>
<tr>
<td>6.3.3.2</td>
<td>length</td>
<td>145</td>
</tr>
<tr>
<td>6.3.3.3</td>
<td>max_size</td>
<td>145</td>
</tr>
<tr>
<td>6.3.3.4</td>
<td>resize</td>
<td>145</td>
</tr>
<tr>
<td>6.3.3.5</td>
<td>capacity</td>
<td>146</td>
</tr>
<tr>
<td>6.3.3.6</td>
<td>reserve</td>
<td>146</td>
</tr>
<tr>
<td>6.3.3.7</td>
<td>clear</td>
<td>146</td>
</tr>
<tr>
<td>6.3.3.8</td>
<td>empty</td>
<td>146</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Element Access</td>
<td>147</td>
</tr>
<tr>
<td>6.3.4.1</td>
<td>operator[]</td>
<td>147</td>
</tr>
<tr>
<td>6.3.4.2</td>
<td>at</td>
<td>147</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Modifiers</td>
<td>147</td>
</tr>
<tr>
<td>6.3.5.1</td>
<td>operator+=</td>
<td>147</td>
</tr>
<tr>
<td>6.3.5.2</td>
<td>append</td>
<td>148</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.3.5.3</td>
<td>assign</td>
<td>148</td>
</tr>
<tr>
<td>6.3.5.4</td>
<td>insert</td>
<td>149</td>
</tr>
<tr>
<td>6.3.5.5</td>
<td>erase</td>
<td>149</td>
</tr>
<tr>
<td>6.3.5.6</td>
<td>replace</td>
<td>150</td>
</tr>
<tr>
<td>6.3.5.7</td>
<td>copy</td>
<td>150</td>
</tr>
<tr>
<td>6.3.5.8</td>
<td>swap</td>
<td>151</td>
</tr>
<tr>
<td>6.3.6</td>
<td>String Operations</td>
<td></td>
</tr>
<tr>
<td>6.3.6.1</td>
<td>c_str</td>
<td>151</td>
</tr>
<tr>
<td>6.3.6.2</td>
<td>data</td>
<td>151</td>
</tr>
<tr>
<td>6.3.6.3</td>
<td>get_allocator</td>
<td>151</td>
</tr>
<tr>
<td>6.3.6.4</td>
<td>find</td>
<td>152</td>
</tr>
<tr>
<td>6.3.6.5</td>
<td>rfind</td>
<td>152</td>
</tr>
<tr>
<td>6.3.6.6</td>
<td>find_first_of</td>
<td>152</td>
</tr>
<tr>
<td>6.3.6.7</td>
<td>find_last_of</td>
<td>153</td>
</tr>
<tr>
<td>6.3.6.8</td>
<td>find_first_not_of</td>
<td>153</td>
</tr>
<tr>
<td>6.3.6.9</td>
<td>find_last_not_of</td>
<td>154</td>
</tr>
<tr>
<td>6.3.6.10</td>
<td>substr</td>
<td>154</td>
</tr>
<tr>
<td>6.3.6.11</td>
<td>compare</td>
<td>155</td>
</tr>
<tr>
<td>6.3.7</td>
<td>Non-Member Functions and Operators</td>
<td></td>
</tr>
<tr>
<td>6.3.7.1</td>
<td>operator+</td>
<td>155</td>
</tr>
<tr>
<td>6.3.7.2</td>
<td>operator==</td>
<td>156</td>
</tr>
<tr>
<td>6.3.7.3</td>
<td>operator!=</td>
<td>157</td>
</tr>
<tr>
<td>6.3.7.4</td>
<td>operator&lt;</td>
<td>157</td>
</tr>
<tr>
<td>6.3.7.5</td>
<td>operator&gt;</td>
<td>158</td>
</tr>
<tr>
<td>6.3.7.6</td>
<td>operator&lt;=</td>
<td>158</td>
</tr>
<tr>
<td>6.3.7.7</td>
<td>operator&gt;=</td>
<td>159</td>
</tr>
<tr>
<td>6.3.7.8</td>
<td>swap</td>
<td>159</td>
</tr>
<tr>
<td>6.3.8</td>
<td>Inserters and extractors</td>
<td></td>
</tr>
<tr>
<td>6.3.8.1</td>
<td>operator&gt;&gt;</td>
<td>160</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.3.8.2</td>
<td>operator&lt;&lt;</td>
<td>160</td>
</tr>
<tr>
<td>6.3.8.3</td>
<td>getline</td>
<td>161</td>
</tr>
<tr>
<td>6.4</td>
<td>Null Terminated Sequence Utilities</td>
<td>161</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Character Support</td>
<td>161</td>
</tr>
<tr>
<td>6.4.2</td>
<td>String Support</td>
<td>162</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Input and Output Manipulations</td>
<td>162</td>
</tr>
</tbody>
</table>

**Chapter 7**  
Localization Library

<p>| 7.1            | Supported Locale Names                    | 165  |
| 7.2            | Strings and Characters in Locale Data Files | 166  |
| 7.2.1          | Character Syntax                          | 167  |
| 7.2.2          | Escape sequences                          | 167  |
| 7.2.3          | Errors                                    | 168  |
| 7.2.4          | String Syntax                             | 168  |
| 7.3            | Locales                                   | 169  |
| 7.3.1          | Class locale                              | 169  |
| 7.3.1.1        | Combined Locale Names                     | 170  |
| 7.3.2          | Locale Types                              | 171  |
| 7.3.2.1        | locale::Category                          | 171  |
| 7.3.2.2        | locale::facet                             | 172  |
| 7.3.2.3        | locale::id                                | 172  |
| 7.3.2.4        | Constructors                              | 173  |
| 7.3.2.5        | destructor                                | 173  |
| 7.3.3          | Locale Members                            | 173  |
| 7.3.3.1        | combine                                   | 174  |
| 7.3.3.2        | name                                      | 174  |
| 7.3.4          | Locale Operators                          | 174  |
| 7.3.4.1        | operator ==                               | 174  |
| 7.3.4.2        | operator !=                               | 175  |</p>
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.4.3</td>
<td>operator () .................................................................................. 175</td>
<td></td>
</tr>
<tr>
<td>7.3.5</td>
<td>Locale Static Members ................................................................. 175</td>
<td></td>
</tr>
<tr>
<td>7.3.5.1</td>
<td>global .......................................................................................... 175</td>
<td></td>
</tr>
<tr>
<td>7.3.5.2</td>
<td>classic ....................................................................................... 176</td>
<td></td>
</tr>
<tr>
<td>7.3.6</td>
<td>Locale Globals .............................................................................. 176</td>
<td></td>
</tr>
<tr>
<td>7.3.6.1</td>
<td>use_facet .................................................................................... 176</td>
<td></td>
</tr>
<tr>
<td>7.3.6.2</td>
<td>has_facet .................................................................................... 177</td>
<td></td>
</tr>
<tr>
<td>7.3.7</td>
<td>Convenience Interfaces ................................................................... 177</td>
<td></td>
</tr>
<tr>
<td>7.3.8</td>
<td>Character Classification .............................................................. 177</td>
<td></td>
</tr>
<tr>
<td>7.3.9</td>
<td>Character Conversions .................................................................... 177</td>
<td></td>
</tr>
<tr>
<td>7.3.9.1</td>
<td>toupper ....................................................................................... 178</td>
<td></td>
</tr>
<tr>
<td>7.3.9.2</td>
<td>tolower ....................................................................................... 178</td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Standard Locale Categories .......................................................... 178</td>
<td></td>
</tr>
<tr>
<td>7.4.1</td>
<td>The Ctype Category .......................................................................... 179</td>
<td></td>
</tr>
<tr>
<td>7.4.1.1</td>
<td>Template Class Ctype ....................................................................... 179</td>
<td></td>
</tr>
<tr>
<td>7.4.1.1.1</td>
<td>is ............................................................................................... 180</td>
<td></td>
</tr>
<tr>
<td>7.4.1.1.2</td>
<td>scan_is ....................................................................................... 180</td>
<td></td>
</tr>
<tr>
<td>7.4.1.1.3</td>
<td>scan_not ...................................................................................... 180</td>
<td></td>
</tr>
<tr>
<td>7.4.1.1.4</td>
<td>toupper ....................................................................................... 181</td>
<td></td>
</tr>
<tr>
<td>7.4.1.1.5</td>
<td>tolower ....................................................................................... 181</td>
<td></td>
</tr>
<tr>
<td>7.4.1.1.6</td>
<td>widen ........................................................................................... 181</td>
<td></td>
</tr>
<tr>
<td>7.4.1.1.7</td>
<td>narrow ........................................................................................ 181</td>
<td></td>
</tr>
<tr>
<td>7.4.1.2</td>
<td>ctype Virtual Functions ................................................................... 182</td>
<td></td>
</tr>
<tr>
<td>7.4.1.2.1</td>
<td>do_is ......................................................................................... 182</td>
<td></td>
</tr>
<tr>
<td>7.4.1.2.2</td>
<td>do_scan_is .................................................................................. 182</td>
<td></td>
</tr>
<tr>
<td>7.4.1.2.3</td>
<td>do_scan_not ................................................................................ 182</td>
<td></td>
</tr>
<tr>
<td>7.4.1.2.4</td>
<td>do_toupper .................................................................................. 182</td>
<td></td>
</tr>
<tr>
<td>7.4.1.2.5</td>
<td>do_tolower .................................................................................. 183</td>
<td></td>
</tr>
<tr>
<td>7.4.1.2.6</td>
<td>do_widen ..................................................................................... 183</td>
<td></td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.4.1.2.7</td>
<td>do_narrow</td>
<td>183</td>
</tr>
<tr>
<td>7.4.1.3</td>
<td>Template class ctype_byname</td>
<td>183</td>
</tr>
<tr>
<td>7.4.1.3.1</td>
<td>ctype_byname Constructor</td>
<td>183</td>
</tr>
<tr>
<td>7.4.1.3.2</td>
<td>Classification</td>
<td>184</td>
</tr>
<tr>
<td>7.4.1.3.3</td>
<td>Case Transformation</td>
<td>185</td>
</tr>
<tr>
<td>7.4.1.4</td>
<td>ctype Specializations</td>
<td>188</td>
</tr>
<tr>
<td>7.4.1.5</td>
<td>Specialized Ctype Constructor and Destructor</td>
<td>189</td>
</tr>
<tr>
<td>7.4.1.5.1</td>
<td>Constructor</td>
<td>189</td>
</tr>
<tr>
<td>7.4.1.5.2</td>
<td>destructor</td>
<td>189</td>
</tr>
<tr>
<td>7.4.1.5.3</td>
<td>Specialized Ctype Members</td>
<td>189</td>
</tr>
<tr>
<td>7.4.1.6</td>
<td>ctype&lt;Char&gt; Static Members</td>
<td>190</td>
</tr>
<tr>
<td>7.4.1.6.1</td>
<td>classic_table</td>
<td>190</td>
</tr>
<tr>
<td>7.4.1.7</td>
<td>Class ctype_byname&lt;char&gt;</td>
<td>190</td>
</tr>
<tr>
<td>7.4.1.7.1</td>
<td>ctype_byname&lt;char&gt; Constructor</td>
<td>191</td>
</tr>
<tr>
<td>7.4.1.8</td>
<td>Template Class Codecvt</td>
<td>191</td>
</tr>
<tr>
<td>7.4.1.8.1</td>
<td>codecvt Members</td>
<td>191</td>
</tr>
<tr>
<td>7.4.1.8.1.1</td>
<td>out</td>
<td>191</td>
</tr>
<tr>
<td>7.4.1.8.1.2</td>
<td>unshift</td>
<td>191</td>
</tr>
<tr>
<td>7.4.1.8.1.3</td>
<td>in</td>
<td>192</td>
</tr>
<tr>
<td>7.4.1.8.1.4</td>
<td>always_noconv</td>
<td>192</td>
</tr>
<tr>
<td>7.4.1.8.1.5</td>
<td>length</td>
<td>192</td>
</tr>
<tr>
<td>7.4.1.8.1.6</td>
<td>max_length</td>
<td>192</td>
</tr>
<tr>
<td>7.4.1.8.1.7</td>
<td>codecvt Virtual Functions</td>
<td>193</td>
</tr>
<tr>
<td>7.4.1.9</td>
<td>Template Class Codecvt_byname</td>
<td>194</td>
</tr>
<tr>
<td>7.4.1.10</td>
<td>Codecvt_byname Keywords</td>
<td>195</td>
</tr>
<tr>
<td>7.4.1.10.1</td>
<td>noconv</td>
<td>195</td>
</tr>
<tr>
<td>7.4.1.10.2</td>
<td>UCS-2</td>
<td>195</td>
</tr>
<tr>
<td>7.4.1.10.3</td>
<td>JIS</td>
<td>195</td>
</tr>
<tr>
<td>7.4.1.10.4</td>
<td>Shift-JIS</td>
<td>196</td>
</tr>
</tbody>
</table>
## Section number | Title | Page
---|---|---
7.4.1.10.5 | EUC | 196
7.4.1.10.6 | UTF-8 | 196
7.4.1.11 | Extending codecvt by derivation | 197
7.4.2 | The Numeric Category | 198
7.4.2.1 | Template Class Num_get | 198
7.4.2.2 | Num_get Members | 198
7.4.2.2.1 | get | 198
7.4.2.2.2 | Num_get Virtual Functions | 199
7.4.2.3 | Num_put Members | 200
7.4.2.3.1 | put | 200
7.4.2.3.2 | Num_put Virtual Functions | 200
7.4.2.4 | The Numeric Punctuation Facet | 200
7.4.2.4.1 | Numpunct Members | 201
7.4.2.4.1.1 | decimal_point | 201
7.4.2.4.1.2 | thousands_sep | 201
7.4.2.4.1.3 | grouping | 201
7.4.2.4.1.4 | truename | 201
7.4.2.4.1.5 | falsename | 202
7.4.2.4.1.6 | numpunct virtual functions | 202
7.4.2.4.1.7 | Template Class Numpunct_byname | 202
7.4.2.4.1.8 | Numeric_wide | 205
7.4.2.5 | Extending numpunct by derivation | 205
7.4.3 | The Collate Category | 206
7.4.3.1 | Collate Members | 206
7.4.3.1.1 | compare | 206
7.4.3.1.2 | transform | 207
7.4.3.1.3 | hash | 207
7.4.3.1.4 | collate Virtual Functions | 207
7.4.3.2 | Template Class Collate_byname | 208
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.3.2.1</td>
<td>Collate Data Section</td>
<td>208</td>
</tr>
<tr>
<td>7.4.3.2.2</td>
<td>Rule Format</td>
<td>209</td>
</tr>
<tr>
<td>7.4.3.2.3</td>
<td>Text-Argument:</td>
<td>209</td>
</tr>
<tr>
<td>7.4.3.2.4</td>
<td>Modifier:</td>
<td>209</td>
</tr>
<tr>
<td>7.4.3.2.5</td>
<td>Relation:</td>
<td>209</td>
</tr>
<tr>
<td>7.4.3.2.6</td>
<td>Reset:</td>
<td>209</td>
</tr>
<tr>
<td>7.4.3.2.7</td>
<td>Relationals</td>
<td>210</td>
</tr>
<tr>
<td>7.4.3.2.8</td>
<td>French collation</td>
<td>211</td>
</tr>
<tr>
<td>7.4.3.2.9</td>
<td>Contraction</td>
<td>211</td>
</tr>
<tr>
<td>7.4.3.2.10</td>
<td>Expansion</td>
<td>211</td>
</tr>
<tr>
<td>7.4.3.2.11</td>
<td>Ignorable Characters</td>
<td>212</td>
</tr>
<tr>
<td>7.4.3.3</td>
<td>Extending collate by derivation</td>
<td>214</td>
</tr>
<tr>
<td>7.4.4</td>
<td>The Time Category</td>
<td>217</td>
</tr>
<tr>
<td>7.4.4.1</td>
<td>Time_get Members</td>
<td>217</td>
</tr>
<tr>
<td>7.4.4.2</td>
<td>Time_get Virtual Functions</td>
<td>218</td>
</tr>
<tr>
<td>7.4.4.3</td>
<td>Format Parsing</td>
<td>220</td>
</tr>
<tr>
<td>7.4.4.4</td>
<td>ISO 8601 week-based year</td>
<td>225</td>
</tr>
<tr>
<td>7.4.4.5</td>
<td>Template Class Time_get_byname</td>
<td>225</td>
</tr>
<tr>
<td>7.4.4.6</td>
<td>Time_put Members</td>
<td>226</td>
</tr>
<tr>
<td>7.4.4.7</td>
<td>Time_put Virtual Functions</td>
<td>226</td>
</tr>
<tr>
<td>7.4.4.8</td>
<td>Template Class Time_put_byname Synopsis</td>
<td>227</td>
</tr>
<tr>
<td>7.4.4.9</td>
<td>Extending The Behavior Of The Time Facets</td>
<td>227</td>
</tr>
<tr>
<td>7.4.4.10</td>
<td>Extending locale by using named locale facilities</td>
<td>227</td>
</tr>
<tr>
<td>7.4.4.10.1</td>
<td>abrev_weekday</td>
<td>228</td>
</tr>
<tr>
<td>7.4.4.10.2</td>
<td>weekday</td>
<td>229</td>
</tr>
<tr>
<td>7.4.4.10.3</td>
<td>abrev_monthname</td>
<td>229</td>
</tr>
<tr>
<td>7.4.4.10.4</td>
<td>monthname</td>
<td>229</td>
</tr>
<tr>
<td>7.4.4.10.5</td>
<td>date_time</td>
<td>229</td>
</tr>
<tr>
<td>7.4.4.10.6</td>
<td>am_pm</td>
<td>230</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>7.4.4.10.7</td>
<td>time_12hour</td>
<td>230</td>
</tr>
<tr>
<td>7.4.4.10.8</td>
<td>date</td>
<td>230</td>
</tr>
<tr>
<td>7.4.4.10.9</td>
<td>time</td>
<td>230</td>
</tr>
<tr>
<td>7.4.4.10.10</td>
<td>time_zone</td>
<td>231</td>
</tr>
<tr>
<td>7.4.4.10.11</td>
<td>utc_offset</td>
<td>231</td>
</tr>
<tr>
<td>7.4.4.10.12</td>
<td>default_century</td>
<td>231</td>
</tr>
<tr>
<td>7.4.4.11</td>
<td>Extending by derivation</td>
<td>235</td>
</tr>
<tr>
<td>7.4.4.12</td>
<td>Timepunct_byname</td>
<td>237</td>
</tr>
<tr>
<td>7.4.5</td>
<td>The Monetary Category</td>
<td>240</td>
</tr>
<tr>
<td>7.4.5.1</td>
<td>A sample Money class</td>
<td>241</td>
</tr>
<tr>
<td>7.4.5.2</td>
<td>Template Class Money_get</td>
<td>246</td>
</tr>
<tr>
<td>7.4.5.2.1</td>
<td>Money_get Members</td>
<td>247</td>
</tr>
<tr>
<td>7.4.5.2.1.1</td>
<td>get</td>
<td>247</td>
</tr>
<tr>
<td>7.4.5.2.1.2</td>
<td>Money_get Virtual Functions</td>
<td>248</td>
</tr>
<tr>
<td>7.4.5.3</td>
<td>Template Class Money_put</td>
<td>248</td>
</tr>
<tr>
<td>7.4.5.3.1</td>
<td>Money_put Members</td>
<td>249</td>
</tr>
<tr>
<td>7.4.5.3.1.1</td>
<td>put</td>
<td>249</td>
</tr>
<tr>
<td>7.4.5.3.1.2</td>
<td>Money_put Virtual Functions</td>
<td>249</td>
</tr>
<tr>
<td>7.4.5.4</td>
<td>Class Moneypunct</td>
<td>250</td>
</tr>
<tr>
<td>7.4.5.4.1</td>
<td>Moneypunct Members</td>
<td>251</td>
</tr>
<tr>
<td>7.4.5.4.1.1</td>
<td>decimal_point</td>
<td>251</td>
</tr>
<tr>
<td>7.4.5.4.1.2</td>
<td>thousands_sep</td>
<td>251</td>
</tr>
<tr>
<td>7.4.5.4.1.3</td>
<td>grouping</td>
<td>252</td>
</tr>
<tr>
<td>7.4.5.4.1.4</td>
<td>curr_symbol</td>
<td>252</td>
</tr>
<tr>
<td>7.4.5.4.1.5</td>
<td>positive_sign</td>
<td>252</td>
</tr>
<tr>
<td>7.4.5.4.1.6</td>
<td>negative_sign</td>
<td>252</td>
</tr>
<tr>
<td>7.4.5.4.1.7</td>
<td>frac_digits</td>
<td>253</td>
</tr>
<tr>
<td>7.4.5.4.1.8</td>
<td>pos_format</td>
<td>253</td>
</tr>
<tr>
<td>7.4.5.4.1.9</td>
<td>neg_format</td>
<td>253</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>7.4.5.4.1.10</td>
<td>Moneypunct Virtual Functions</td>
<td>254</td>
</tr>
<tr>
<td>7.4.5.5</td>
<td>Extending moneypunct by derivation</td>
<td>255</td>
</tr>
<tr>
<td>7.4.5.6</td>
<td>Template Class Moneypunct_byname</td>
<td>256</td>
</tr>
<tr>
<td>7.4.5.7</td>
<td>Data file syntax</td>
<td>257</td>
</tr>
<tr>
<td>7.4.5.7.1</td>
<td>decimal_point</td>
<td>258</td>
</tr>
<tr>
<td>7.4.5.7.2</td>
<td>thousands_sep</td>
<td>258</td>
</tr>
<tr>
<td>7.4.5.7.3</td>
<td>grouping</td>
<td>258</td>
</tr>
<tr>
<td>7.4.5.7.4</td>
<td>curr_symbol</td>
<td>259</td>
</tr>
<tr>
<td>7.4.5.7.5</td>
<td>positive_sign</td>
<td>259</td>
</tr>
<tr>
<td>7.4.5.7.6</td>
<td>negative_sign</td>
<td>259</td>
</tr>
<tr>
<td>7.4.5.7.7</td>
<td>frac_digits</td>
<td>260</td>
</tr>
<tr>
<td>7.4.5.7.8</td>
<td>pos_format / neg_format</td>
<td>260</td>
</tr>
<tr>
<td>7.4.6</td>
<td>The Message Retrieval Category</td>
<td>261</td>
</tr>
<tr>
<td>7.4.6.1</td>
<td>Messages Members</td>
<td>262</td>
</tr>
<tr>
<td>7.4.6.1.1</td>
<td>open</td>
<td>262</td>
</tr>
<tr>
<td>7.4.6.1.2</td>
<td>get</td>
<td>263</td>
</tr>
<tr>
<td>7.4.6.1.3</td>
<td>close</td>
<td>263</td>
</tr>
<tr>
<td>7.4.6.1.4</td>
<td>Messages Virtual Functions</td>
<td>263</td>
</tr>
<tr>
<td>7.4.6.2</td>
<td>EWL C++ implementation of messages</td>
<td>264</td>
</tr>
<tr>
<td>7.4.6.3</td>
<td>Template Class Messages_byname Synopsis</td>
<td>266</td>
</tr>
<tr>
<td>7.4.6.4</td>
<td>Extending messages by derivation</td>
<td>267</td>
</tr>
<tr>
<td>7.4.7</td>
<td>Program-defined Facets</td>
<td>269</td>
</tr>
<tr>
<td>7.5</td>
<td>C Library Locales</td>
<td>269</td>
</tr>
<tr>
<td>8.1</td>
<td>Container Requirements</td>
<td>271</td>
</tr>
<tr>
<td>8.1.1</td>
<td>All containers must meet basic requirements</td>
<td>271</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Unless specified containers meet these requirements</td>
<td>272</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Sequences Requirements</td>
<td>272</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.1.3.1</td>
<td>8.1.3.1 Additional Requirements</td>
<td>272</td>
</tr>
<tr>
<td>8.1.4</td>
<td>8.1.4 Associative Containers Requirements</td>
<td>273</td>
</tr>
<tr>
<td>8.2</td>
<td>8.2 Sequences</td>
<td>274</td>
</tr>
<tr>
<td>8.2.1</td>
<td>8.2.1 Template Class Deque</td>
<td>274</td>
</tr>
<tr>
<td>8.2.1.1</td>
<td>8.2.1.1 Constructors</td>
<td>274</td>
</tr>
<tr>
<td>8.2.1.2</td>
<td>8.2.1.2 assign</td>
<td>275</td>
</tr>
<tr>
<td>8.2.1.3</td>
<td>8.2.1.3 resize</td>
<td>275</td>
</tr>
<tr>
<td>8.2.1.4</td>
<td>8.2.1.4 insert</td>
<td>275</td>
</tr>
<tr>
<td>8.2.1.5</td>
<td>8.2.1.5 erase</td>
<td>276</td>
</tr>
<tr>
<td>8.2.1.6</td>
<td>8.2.1.6 swap</td>
<td>276</td>
</tr>
<tr>
<td>8.2.2</td>
<td>8.2.2 Template Class List</td>
<td>276</td>
</tr>
<tr>
<td>8.2.2.1</td>
<td>8.2.2.1 Constructors</td>
<td>276</td>
</tr>
<tr>
<td>8.2.2.2</td>
<td>8.2.2.2 assign</td>
<td>277</td>
</tr>
<tr>
<td>8.2.2.3</td>
<td>8.2.2.3 resize</td>
<td>277</td>
</tr>
<tr>
<td>8.2.2.4</td>
<td>8.2.2.4 insert</td>
<td>277</td>
</tr>
<tr>
<td>8.2.2.5</td>
<td>8.2.2.5 push_front</td>
<td>278</td>
</tr>
<tr>
<td>8.2.2.6</td>
<td>8.2.2.6 push_back</td>
<td>278</td>
</tr>
<tr>
<td>8.2.2.7</td>
<td>8.2.2.7 erase</td>
<td>278</td>
</tr>
<tr>
<td>8.2.2.8</td>
<td>8.2.2.8 pop_front</td>
<td>278</td>
</tr>
<tr>
<td>8.2.2.9</td>
<td>8.2.2.9 pop_back</td>
<td>278</td>
</tr>
<tr>
<td>8.2.2.10</td>
<td>8.2.2.10 clear</td>
<td>279</td>
</tr>
<tr>
<td>8.2.2.11</td>
<td>8.2.2.11 splice</td>
<td>279</td>
</tr>
<tr>
<td>8.2.2.12</td>
<td>8.2.2.12 remove</td>
<td>279</td>
</tr>
<tr>
<td>8.2.2.13</td>
<td>8.2.2.13 remove_if</td>
<td>279</td>
</tr>
<tr>
<td>8.2.2.14</td>
<td>8.2.2.14 unique</td>
<td>280</td>
</tr>
<tr>
<td>8.2.2.15</td>
<td>8.2.2.15 merge</td>
<td>280</td>
</tr>
<tr>
<td>8.2.2.16</td>
<td>8.2.2.16 reverse</td>
<td>280</td>
</tr>
<tr>
<td>8.2.2.17</td>
<td>8.2.2.17 sort</td>
<td>280</td>
</tr>
<tr>
<td>8.2.2.18</td>
<td>8.2.2.18 swap</td>
<td>281</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Container Adaptors</td>
<td>281</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Template Class Queue</td>
<td>281</td>
</tr>
<tr>
<td>8.2.4.1</td>
<td>operator ==</td>
<td>281</td>
</tr>
<tr>
<td>8.2.4.2</td>
<td>operator &lt;</td>
<td>282</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Template Class Priority_queue</td>
<td>282</td>
</tr>
<tr>
<td>8.2.5.1</td>
<td>Constructors</td>
<td>282</td>
</tr>
<tr>
<td>8.2.5.2</td>
<td>push</td>
<td>282</td>
</tr>
<tr>
<td>8.2.5.3</td>
<td>pop</td>
<td>283</td>
</tr>
<tr>
<td>8.2.6</td>
<td>Template Class Stack</td>
<td>283</td>
</tr>
<tr>
<td>8.2.6.1</td>
<td>Public Member Functions</td>
<td>283</td>
</tr>
<tr>
<td>8.2.6.1.1</td>
<td>Constructors</td>
<td>283</td>
</tr>
<tr>
<td>8.2.6.1.2</td>
<td>empty</td>
<td>283</td>
</tr>
<tr>
<td>8.2.6.1.3</td>
<td>size</td>
<td>284</td>
</tr>
<tr>
<td>8.2.6.1.4</td>
<td>top</td>
<td>284</td>
</tr>
<tr>
<td>8.2.6.1.5</td>
<td>push</td>
<td>284</td>
</tr>
<tr>
<td>8.2.6.1.6</td>
<td>pop</td>
<td>284</td>
</tr>
<tr>
<td>8.2.7</td>
<td>Template Class Vector</td>
<td>284</td>
</tr>
<tr>
<td>8.2.7.1</td>
<td>Constructors</td>
<td>285</td>
</tr>
<tr>
<td>8.2.7.2</td>
<td>assign</td>
<td>285</td>
</tr>
<tr>
<td>8.2.7.3</td>
<td>capacity</td>
<td>285</td>
</tr>
<tr>
<td>8.2.7.4</td>
<td>resize</td>
<td>286</td>
</tr>
<tr>
<td>8.2.7.5</td>
<td>insert</td>
<td>286</td>
</tr>
<tr>
<td>8.2.7.6</td>
<td>erase</td>
<td>286</td>
</tr>
<tr>
<td>8.2.7.7</td>
<td>swap</td>
<td>286</td>
</tr>
<tr>
<td>8.2.8</td>
<td>Class Vector&lt;bool&gt;</td>
<td>287</td>
</tr>
<tr>
<td>8.3</td>
<td>Associative Containers</td>
<td>287</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Template Class Map</td>
<td>287</td>
</tr>
<tr>
<td>8.3.1.1</td>
<td>Constructors</td>
<td>287</td>
</tr>
<tr>
<td>8.3.1.2</td>
<td>Map Element Access</td>
<td>288</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.3.1.2.1</td>
<td>operator []</td>
<td>288</td>
</tr>
<tr>
<td>8.3.1.3</td>
<td>Map Operations</td>
<td>288</td>
</tr>
<tr>
<td>8.3.1.3.1</td>
<td>find</td>
<td>288</td>
</tr>
<tr>
<td>8.3.1.3.2</td>
<td>lower_bound</td>
<td>288</td>
</tr>
<tr>
<td>8.3.1.3.3</td>
<td>upper_bound</td>
<td>289</td>
</tr>
<tr>
<td>8.3.1.3.4</td>
<td>equal_range</td>
<td>289</td>
</tr>
<tr>
<td>8.3.1.4</td>
<td>Map Specialized Algorithms</td>
<td>289</td>
</tr>
<tr>
<td>8.3.1.4.1</td>
<td>swap</td>
<td>289</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Template Class Multimap</td>
<td>290</td>
</tr>
<tr>
<td>8.3.2.1</td>
<td>Constructors</td>
<td>290</td>
</tr>
<tr>
<td>8.3.2.2</td>
<td>Multimap Operations</td>
<td>290</td>
</tr>
<tr>
<td>8.3.2.2.1</td>
<td>find</td>
<td>290</td>
</tr>
<tr>
<td>8.3.2.2.2</td>
<td>lower_bound</td>
<td>291</td>
</tr>
<tr>
<td>8.3.2.2.3</td>
<td>equal_range</td>
<td>291</td>
</tr>
<tr>
<td>8.3.2.3</td>
<td>Multimap Specialized Algorithms</td>
<td>291</td>
</tr>
<tr>
<td>8.3.2.3.1</td>
<td>swap</td>
<td>291</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Template Class Set</td>
<td>292</td>
</tr>
<tr>
<td>8.3.3.1</td>
<td>Constructors</td>
<td>292</td>
</tr>
<tr>
<td>8.3.3.2</td>
<td>Set Specialized Algorithms</td>
<td>292</td>
</tr>
<tr>
<td>8.3.3.2.1</td>
<td>swap</td>
<td>292</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Template Class Multiset</td>
<td>293</td>
</tr>
<tr>
<td>8.3.4.1</td>
<td>Constructors</td>
<td>293</td>
</tr>
<tr>
<td>8.3.4.2</td>
<td>Multiset Specialized Algorithms</td>
<td>293</td>
</tr>
<tr>
<td>8.3.4.2.1</td>
<td>swap</td>
<td>293</td>
</tr>
<tr>
<td>8.3.5</td>
<td>Template Class Bitset</td>
<td>293</td>
</tr>
<tr>
<td>8.3.5.1</td>
<td>Constructors</td>
<td>294</td>
</tr>
<tr>
<td>8.3.5.2</td>
<td>Bitset Members</td>
<td>294</td>
</tr>
<tr>
<td>8.3.5.2.1</td>
<td>operator &amp;=</td>
<td>294</td>
</tr>
<tr>
<td>8.3.5.2.2</td>
<td>operator</td>
<td>=</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.3.5.2.3</td>
<td>operator ^=</td>
<td>295</td>
</tr>
<tr>
<td>8.3.5.2.4</td>
<td>operator &lt;&lt;=</td>
<td>295</td>
</tr>
<tr>
<td>8.3.5.2.5</td>
<td>operator &gt;&gt;=</td>
<td>295</td>
</tr>
<tr>
<td>8.3.5.2.6</td>
<td>Set</td>
<td>296</td>
</tr>
<tr>
<td>8.3.5.2.7</td>
<td>reset</td>
<td>296</td>
</tr>
<tr>
<td>8.3.5.2.8</td>
<td>operator ~</td>
<td>296</td>
</tr>
<tr>
<td>8.3.5.2.9</td>
<td>flip</td>
<td>297</td>
</tr>
<tr>
<td>8.3.5.2.10</td>
<td>to_ulong</td>
<td>297</td>
</tr>
<tr>
<td>8.3.5.2.11</td>
<td>to_string</td>
<td>297</td>
</tr>
<tr>
<td>8.3.5.2.12</td>
<td>count</td>
<td>297</td>
</tr>
<tr>
<td>8.3.5.2.13</td>
<td>size</td>
<td>298</td>
</tr>
<tr>
<td>8.3.5.2.14</td>
<td>operator ==</td>
<td>298</td>
</tr>
<tr>
<td>8.3.5.2.15</td>
<td>operator !=</td>
<td>298</td>
</tr>
<tr>
<td>8.3.5.2.16</td>
<td>test</td>
<td>298</td>
</tr>
<tr>
<td>8.3.5.2.17</td>
<td>any</td>
<td>299</td>
</tr>
<tr>
<td>8.3.5.2.18</td>
<td>none</td>
<td>299</td>
</tr>
<tr>
<td>8.3.5.2.19</td>
<td>operator &lt;&lt;=</td>
<td>299</td>
</tr>
<tr>
<td>8.3.5.2.20</td>
<td>operator &gt;&gt;=</td>
<td>299</td>
</tr>
<tr>
<td>8.3.5.3</td>
<td>Bitset Operators</td>
<td>300</td>
</tr>
<tr>
<td>8.3.5.3.1</td>
<td>operator &amp;</td>
<td>300</td>
</tr>
<tr>
<td>8.3.5.3.2</td>
<td>operator</td>
<td></td>
</tr>
<tr>
<td>8.3.5.3.3</td>
<td>operator ^</td>
<td>300</td>
</tr>
<tr>
<td>8.3.5.3.4</td>
<td>operator &gt;&gt;=</td>
<td>301</td>
</tr>
<tr>
<td>8.3.5.3.5</td>
<td>operator &lt;&lt;=</td>
<td>301</td>
</tr>
</tbody>
</table>

Chapter 9
Iterators Library

9.1  Requirements                                           | 303 |
9.1.1 Input Iterators                                       | 303 |
9.1.2 Output Iterators                                      | 304 |
Chapter 10
Algorithms Library

10.1 Header algorithm......................................................................................................................... 323
10.1.1 Non-modifying Sequence Operations......................................................................................... 323
10.1.1.1 for_each.................................................................................................................................. 323
10.1.1.2 find..................................................................................................................................... 324
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1.1.3</td>
<td>find_if</td>
<td>324</td>
</tr>
<tr>
<td>10.1.1.4</td>
<td>find_end</td>
<td>324</td>
</tr>
<tr>
<td>10.1.1.5</td>
<td>find_first_of</td>
<td>325</td>
</tr>
<tr>
<td>10.1.1.6</td>
<td>adjacent_find</td>
<td>325</td>
</tr>
<tr>
<td>10.1.1.7</td>
<td>count</td>
<td>326</td>
</tr>
<tr>
<td>10.1.1.8</td>
<td>count_if</td>
<td>326</td>
</tr>
<tr>
<td>10.1.1.9</td>
<td>mismatch</td>
<td>326</td>
</tr>
<tr>
<td>10.1.1.10</td>
<td>equal</td>
<td>327</td>
</tr>
<tr>
<td>10.1.1.11</td>
<td>search</td>
<td>327</td>
</tr>
<tr>
<td>10.1.1.12</td>
<td>search_n</td>
<td>328</td>
</tr>
<tr>
<td>10.1.2</td>
<td>Mutating Sequence Operators</td>
<td>328</td>
</tr>
<tr>
<td>10.1.2.1</td>
<td>copy</td>
<td>328</td>
</tr>
<tr>
<td>10.1.2.2</td>
<td>copy_backward</td>
<td>328</td>
</tr>
<tr>
<td>10.1.2.3</td>
<td>swap</td>
<td>329</td>
</tr>
<tr>
<td>10.1.2.4</td>
<td>swap_ranges</td>
<td>329</td>
</tr>
<tr>
<td>10.1.2.5</td>
<td>iter_swap</td>
<td>329</td>
</tr>
<tr>
<td>10.1.2.6</td>
<td>transform</td>
<td>330</td>
</tr>
<tr>
<td>10.1.2.7</td>
<td>replace</td>
<td>330</td>
</tr>
<tr>
<td>10.1.2.8</td>
<td>replace_copy</td>
<td>330</td>
</tr>
<tr>
<td>10.1.2.9</td>
<td>replace_copy_if</td>
<td>331</td>
</tr>
<tr>
<td>10.1.2.10</td>
<td>fill</td>
<td>331</td>
</tr>
<tr>
<td>10.1.2.11</td>
<td>fill_n</td>
<td>331</td>
</tr>
<tr>
<td>10.1.2.12</td>
<td>generate</td>
<td>332</td>
</tr>
<tr>
<td>10.1.2.13</td>
<td>generate_n</td>
<td>332</td>
</tr>
<tr>
<td>10.1.2.14</td>
<td>remove</td>
<td>332</td>
</tr>
<tr>
<td>10.1.2.15</td>
<td>remove_if</td>
<td>333</td>
</tr>
<tr>
<td>10.1.2.16</td>
<td>remove_copy</td>
<td>333</td>
</tr>
<tr>
<td>10.1.2.17</td>
<td>remove_copy_if</td>
<td>333</td>
</tr>
<tr>
<td>10.1.2.18</td>
<td>unique</td>
<td>334</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------</td>
<td>------</td>
</tr>
<tr>
<td>10.1.2.19</td>
<td>unique_copy</td>
<td>334</td>
</tr>
<tr>
<td>10.1.2.20</td>
<td>reverse</td>
<td>334</td>
</tr>
<tr>
<td>10.1.2.21</td>
<td>reverse_copy</td>
<td>335</td>
</tr>
<tr>
<td>10.1.2.22</td>
<td>rotate</td>
<td>335</td>
</tr>
<tr>
<td>10.1.2.23</td>
<td>rotate_copy</td>
<td>335</td>
</tr>
<tr>
<td>10.1.2.24</td>
<td>random_shuffle</td>
<td>336</td>
</tr>
<tr>
<td>10.1.2.25</td>
<td>partition</td>
<td>336</td>
</tr>
<tr>
<td>10.1.2.26</td>
<td>stable_partition</td>
<td>336</td>
</tr>
<tr>
<td>10.1.3</td>
<td>Sorting And Related Operations</td>
<td>337</td>
</tr>
<tr>
<td>10.1.3.1</td>
<td>sort</td>
<td>337</td>
</tr>
<tr>
<td>10.1.3.2</td>
<td>stable_sort</td>
<td>337</td>
</tr>
<tr>
<td>10.1.3.3</td>
<td>partial_sort</td>
<td>338</td>
</tr>
<tr>
<td>10.1.3.4</td>
<td>partial_sort_copy</td>
<td>338</td>
</tr>
<tr>
<td>10.1.3.5</td>
<td>nth_element</td>
<td>339</td>
</tr>
<tr>
<td>10.1.3.6</td>
<td>lower_bound</td>
<td>339</td>
</tr>
<tr>
<td>10.1.3.7</td>
<td>upper_bound</td>
<td>339</td>
</tr>
<tr>
<td>10.1.3.8</td>
<td>equal_range</td>
<td>340</td>
</tr>
<tr>
<td>10.1.3.9</td>
<td>binary_search</td>
<td>340</td>
</tr>
<tr>
<td>10.1.3.10</td>
<td>merge</td>
<td>340</td>
</tr>
<tr>
<td>10.1.3.11</td>
<td>inplace_merge</td>
<td>341</td>
</tr>
<tr>
<td>10.1.3.12</td>
<td>includes</td>
<td>341</td>
</tr>
<tr>
<td>10.1.3.13</td>
<td>set_union</td>
<td>342</td>
</tr>
<tr>
<td>10.1.3.14</td>
<td>set_intersection</td>
<td>342</td>
</tr>
<tr>
<td>10.1.3.15</td>
<td>set_difference</td>
<td>343</td>
</tr>
<tr>
<td>10.1.3.16</td>
<td>set_symetric_difference</td>
<td>343</td>
</tr>
<tr>
<td>10.1.3.17</td>
<td>push_heap</td>
<td>344</td>
</tr>
<tr>
<td>10.1.3.18</td>
<td>pop_heap</td>
<td>344</td>
</tr>
<tr>
<td>10.1.3.19</td>
<td>make_heap</td>
<td>344</td>
</tr>
<tr>
<td>10.1.3.20</td>
<td>sort_heap</td>
<td>345</td>
</tr>
</tbody>
</table>
Chapter 11
Numerics Library

11.1 Numeric type requirements.................................................................351
11.2 Numeric arrays......................................................................................352
  11.2.1 Template Class Valarray.................................................................352
    11.2.1.1 Constructors...........................................................................352
    11.2.1.2 Destructor...............................................................................353
    11.2.1.3 Assignment Operator...............................................................353
    11.2.1.4 operator[]..............................................................................354
    11.2.1.5 operator[]..............................................................................354
    11.2.1.6 valarray unary operators.........................................................354
    11.2.1.7 Valarray Computed Assignment..............................................355
  11.2.2 Valarray Member Functions...........................................................356
    11.2.2.1 size.......................................................................................356
    11.2.2.2 sum......................................................................................357
    11.2.2.3 min......................................................................................357
    11.2.2.4 max......................................................................................357
    11.2.2.5 shift.....................................................................................358
    11.2.2.6 cshift....................................................................................358
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.2.2.7</td>
<td>apply</td>
<td>358</td>
</tr>
<tr>
<td>11.2.2.8</td>
<td>resize</td>
<td>358</td>
</tr>
<tr>
<td>11.2.3</td>
<td>Valarray Non-member Operations</td>
<td>359</td>
</tr>
<tr>
<td>11.2.3.1</td>
<td>Valarray Binary Operators</td>
<td>359</td>
</tr>
<tr>
<td>11.2.3.2</td>
<td>Valarray Logical Operators</td>
<td>360</td>
</tr>
<tr>
<td>11.2.4</td>
<td>Non-member logical operations</td>
<td>360</td>
</tr>
<tr>
<td>11.2.4.1</td>
<td>valarray transcendental</td>
<td>361</td>
</tr>
<tr>
<td>11.2.5</td>
<td>Class slice</td>
<td>362</td>
</tr>
<tr>
<td>11.2.5.1</td>
<td>Constructors</td>
<td>362</td>
</tr>
<tr>
<td>11.2.5.2</td>
<td>slice access functions</td>
<td>363</td>
</tr>
<tr>
<td>11.2.5.2.1</td>
<td>start</td>
<td>363</td>
</tr>
<tr>
<td>11.2.5.2.2</td>
<td>size</td>
<td>363</td>
</tr>
<tr>
<td>11.2.5.2.3</td>
<td>stride</td>
<td>363</td>
</tr>
<tr>
<td>11.2.6</td>
<td>Template Class Slice_array</td>
<td>363</td>
</tr>
<tr>
<td>11.2.6.1</td>
<td>Constructors</td>
<td>364</td>
</tr>
<tr>
<td>11.2.6.2</td>
<td>Assignment Operator</td>
<td>364</td>
</tr>
<tr>
<td>11.2.6.3</td>
<td>slice_array computed assignment</td>
<td>364</td>
</tr>
<tr>
<td>11.2.6.4</td>
<td>Slice_array Fill Function</td>
<td>364</td>
</tr>
<tr>
<td>11.2.7</td>
<td>Class Gslice</td>
<td>365</td>
</tr>
<tr>
<td>11.2.7.1</td>
<td>Constructors</td>
<td>365</td>
</tr>
<tr>
<td>11.2.7.2</td>
<td>Gslice Access Functions</td>
<td>365</td>
</tr>
<tr>
<td>11.2.7.2.1</td>
<td>start</td>
<td>365</td>
</tr>
<tr>
<td>11.2.7.2.2</td>
<td>size</td>
<td>365</td>
</tr>
<tr>
<td>11.2.7.2.3</td>
<td>stride</td>
<td>366</td>
</tr>
<tr>
<td>11.2.8</td>
<td>Template Class Gslice_array</td>
<td>366</td>
</tr>
<tr>
<td>11.2.8.1</td>
<td>Constructors</td>
<td>366</td>
</tr>
<tr>
<td>11.2.8.2</td>
<td>Assignment Operators</td>
<td>366</td>
</tr>
<tr>
<td>11.2.8.3</td>
<td>Gslice_array Computed Assignment</td>
<td>367</td>
</tr>
<tr>
<td>11.2.8.4</td>
<td>Fill Function</td>
<td>367</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>11.2.9</td>
<td>Template Class Mask_array</td>
<td>367</td>
</tr>
<tr>
<td>11.2.9.1</td>
<td>Constructors</td>
<td>367</td>
</tr>
<tr>
<td>11.2.9.2</td>
<td>Assignment Operators</td>
<td>368</td>
</tr>
<tr>
<td>11.2.9.3</td>
<td>Mask_array Computed Assignment</td>
<td>368</td>
</tr>
<tr>
<td>11.2.9.4</td>
<td>Mask_array Fill Function</td>
<td>368</td>
</tr>
<tr>
<td>11.2.10</td>
<td>Template Class Indirect_array</td>
<td>369</td>
</tr>
<tr>
<td>11.2.10.1</td>
<td>Constructors</td>
<td>369</td>
</tr>
<tr>
<td>11.2.10.2</td>
<td>Assignment Operators</td>
<td>369</td>
</tr>
<tr>
<td>11.2.10.3</td>
<td>Indirect_array Computed Assignment</td>
<td>369</td>
</tr>
<tr>
<td>11.2.10.4</td>
<td>indirect_array fill function</td>
<td>370</td>
</tr>
<tr>
<td>11.3</td>
<td>Generalized Numeric Operations</td>
<td>370</td>
</tr>
<tr>
<td>11.3.1</td>
<td>Header &lt;numeric&gt;</td>
<td>370</td>
</tr>
<tr>
<td>11.3.1.1</td>
<td>accumulate</td>
<td>370</td>
</tr>
<tr>
<td>11.3.1.2</td>
<td>inner_product</td>
<td>371</td>
</tr>
<tr>
<td>11.3.1.3</td>
<td>partial_sum</td>
<td>371</td>
</tr>
<tr>
<td>11.3.1.4</td>
<td>adjacent_difference</td>
<td>372</td>
</tr>
<tr>
<td>11.4</td>
<td>C Library</td>
<td>372</td>
</tr>
<tr>
<td>11.4.1</td>
<td>&lt;cmath&gt;</td>
<td>372</td>
</tr>
<tr>
<td>11.4.2</td>
<td>&lt;cstdlib&gt;</td>
<td>373</td>
</tr>
</tbody>
</table>

Chapter 12
Complex Class

12.1 Header complex.................................................. 375
12.1.1 _EWL_CX_LIMITED_RANGE..................................... 375
12.1.2 Header <complex> forward declarations.................. 376
12.2 Complex Specializations........................................ 376
12.3 Complex Template Class.......................................... 376
12.3.1 Constructors and Assignments.............................. 377
12.3.1.1 Constructors.................................................. 377
12.3.2 Complex Member Functions.................................... 377
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3.2.1</td>
<td>real</td>
<td>378</td>
</tr>
<tr>
<td>12.3.2.2</td>
<td>imag</td>
<td>378</td>
</tr>
<tr>
<td>12.3.3</td>
<td>Complex Class Operators</td>
<td>378</td>
</tr>
<tr>
<td>12.3.4</td>
<td>Overloaded Operators and Functions</td>
<td>380</td>
</tr>
<tr>
<td>12.3.4.1</td>
<td>Overloaded Complex Operators</td>
<td>380</td>
</tr>
<tr>
<td>12.3.5</td>
<td>Complex Value Operations</td>
<td>382</td>
</tr>
<tr>
<td>12.3.5.1</td>
<td>real</td>
<td>383</td>
</tr>
<tr>
<td>12.3.5.2</td>
<td>imag</td>
<td>383</td>
</tr>
<tr>
<td>12.3.5.3</td>
<td>abs</td>
<td>383</td>
</tr>
<tr>
<td>12.3.5.4</td>
<td>arg</td>
<td>383</td>
</tr>
<tr>
<td>12.3.5.5</td>
<td>norm</td>
<td>384</td>
</tr>
<tr>
<td>12.3.5.6</td>
<td>conj</td>
<td>384</td>
</tr>
<tr>
<td>12.3.5.7</td>
<td>polar</td>
<td>384</td>
</tr>
<tr>
<td>12.3.6</td>
<td>Complex Transcendentals</td>
<td>385</td>
</tr>
<tr>
<td>12.3.6.1</td>
<td>cos</td>
<td>385</td>
</tr>
<tr>
<td>12.3.6.2</td>
<td>cosh</td>
<td>385</td>
</tr>
<tr>
<td>12.3.6.3</td>
<td>exp</td>
<td>386</td>
</tr>
<tr>
<td>12.3.6.4</td>
<td>log</td>
<td>386</td>
</tr>
<tr>
<td>12.3.6.5</td>
<td>log10</td>
<td>386</td>
</tr>
<tr>
<td>12.3.6.6</td>
<td>pow</td>
<td>387</td>
</tr>
<tr>
<td>12.3.6.7</td>
<td>sin</td>
<td>387</td>
</tr>
<tr>
<td>12.3.6.8</td>
<td>sinh</td>
<td>387</td>
</tr>
<tr>
<td>12.3.6.9</td>
<td>sqrt</td>
<td>387</td>
</tr>
<tr>
<td>12.3.6.10</td>
<td>tan</td>
<td>388</td>
</tr>
<tr>
<td>12.3.6.11</td>
<td>tanh</td>
<td>388</td>
</tr>
</tbody>
</table>

Chapter 13
Input and Output Library

13.1 Input and Output Library Summary                                      | 389  |
13.2 Iostreams requirements                                                | 389  |
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.2.1</td>
<td>Definitions</td>
<td>390</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Type requirements</td>
<td>390</td>
</tr>
<tr>
<td>13.2.3</td>
<td>Type SZ_T</td>
<td>390</td>
</tr>
</tbody>
</table>

**Chapter 14**

**Forward Declarations**

14.1 The Streams and String Forward Declarations | 391   
14.2 Header iosfwd | 391   
14.3 Header stringfwd | 391   

**Chapter 15**

**Iostream Objects**

15.1 Header iostream | 395   
15.1.1 Stream Buffering | 395   
15.2 The Standard Input and Output Stream Library | 396   
15.2.1 Narrow stream objects | 396   
15.2.1.1 istream cin | 396   
15.2.1.2 ostream cout | 397   
15.2.1.3 ostream cerr | 397   
15.2.1.4 ostream clog | 397   
15.2.2 Wide stream objects | 398   
15.2.2.1 wistream wcin | 398   
15.2.2.2 wostream wcout | 398   
15.2.2.3 wostream wcerr | 398   
15.2.2.4 wostream wlcog | 399   

**Chapter 16**

**Iostreams Base Classes**

16.1 Header ios | 401   
16.1.1 Template Class fpos | 401   
16.2 Typedef Declarations | 402   
16.3 Class ios_base | 402   
16.3.1 Typedef Declarations | 403   

EWL C++ Library Reference Manual, Rev. 10.x, 02/2014

Freescale Semiconductor, Inc.
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.3.2</td>
<td>Class ios_base::failure...................................................................</td>
<td>403</td>
</tr>
<tr>
<td>16.3.2.1</td>
<td>failure.......................................................................................</td>
<td>403</td>
</tr>
<tr>
<td>16.3.2.2</td>
<td>failure::what............................................................................</td>
<td>403</td>
</tr>
<tr>
<td>16.3.3</td>
<td>Type fmtflags............................................................................</td>
<td>403</td>
</tr>
<tr>
<td>16.3.4</td>
<td>Type iostate..............................................................................</td>
<td>404</td>
</tr>
<tr>
<td>16.3.5</td>
<td>Type openmode...........................................................................</td>
<td>405</td>
</tr>
<tr>
<td>16.3.6</td>
<td>Type seekdir.............................................................................</td>
<td>405</td>
</tr>
<tr>
<td>16.3.7</td>
<td>Class Init...............................................................................</td>
<td>406</td>
</tr>
<tr>
<td>16.3.7.1</td>
<td>Class Init Constructor................................................................</td>
<td>406</td>
</tr>
<tr>
<td>16.3.7.2</td>
<td>Destructor...............................................................................</td>
<td>406</td>
</tr>
<tr>
<td>16.3.8</td>
<td>ios_base fmtflags state functions...........................................</td>
<td>406</td>
</tr>
<tr>
<td>16.3.8.1</td>
<td>flags.....................................................................................</td>
<td>407</td>
</tr>
<tr>
<td>16.3.8.2</td>
<td>setf.......................................................................................</td>
<td>409</td>
</tr>
<tr>
<td>16.3.8.3</td>
<td>unsetf....................................................................................</td>
<td>410</td>
</tr>
<tr>
<td>16.3.8.4</td>
<td>precision..............................................................................</td>
<td>411</td>
</tr>
<tr>
<td>16.3.8.5</td>
<td>width......................................................................................</td>
<td>412</td>
</tr>
<tr>
<td>16.3.9</td>
<td>ios_base locale functions.......................................................</td>
<td>413</td>
</tr>
<tr>
<td>16.3.9.1</td>
<td>imbue.....................................................................................</td>
<td>413</td>
</tr>
<tr>
<td>16.3.9.2</td>
<td>getloc....................................................................................</td>
<td>413</td>
</tr>
<tr>
<td>16.3.10</td>
<td>ios_base storage function.......................................................</td>
<td>414</td>
</tr>
<tr>
<td>16.3.10.1</td>
<td>xalloc..................................................................................</td>
<td>414</td>
</tr>
<tr>
<td>16.3.10.2</td>
<td>iword....................................................................................</td>
<td>414</td>
</tr>
<tr>
<td>16.3.10.3</td>
<td>pword....................................................................................</td>
<td>414</td>
</tr>
<tr>
<td>16.3.10.4</td>
<td>register_callback...................................................................</td>
<td>415</td>
</tr>
<tr>
<td>16.3.10.5</td>
<td>sync_with_stdio.......................................................................</td>
<td>415</td>
</tr>
<tr>
<td>16.3.11</td>
<td>ios_base...............................................................................</td>
<td>416</td>
</tr>
<tr>
<td>16.3.11.1</td>
<td>ios_base Constructor................................................................</td>
<td>416</td>
</tr>
<tr>
<td>16.3.11.2</td>
<td>ios_base Destructor..................................................................</td>
<td>416</td>
</tr>
<tr>
<td>16.4</td>
<td>Template class basic_ios.........................................................</td>
<td>416</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>16.4.1</td>
<td>basic_ios Constructor......................................................................</td>
<td>417</td>
</tr>
<tr>
<td>16.4.2</td>
<td>Destructor..................................................................................</td>
<td>417</td>
</tr>
<tr>
<td>16.4.3</td>
<td>Basic_ios Member Functions................................................................</td>
<td>418</td>
</tr>
<tr>
<td>16.4.3.1</td>
<td>tie.........................................................................................</td>
<td>418</td>
</tr>
<tr>
<td>16.4.3.2</td>
<td>rdbuf.....................................................................................</td>
<td>419</td>
</tr>
<tr>
<td>16.4.3.3</td>
<td>imbue.....................................................................................</td>
<td>420</td>
</tr>
<tr>
<td>16.4.3.4</td>
<td>fill.......................................................................................</td>
<td>421</td>
</tr>
<tr>
<td>16.4.3.5</td>
<td>copyfmt...............................................................................</td>
<td>422</td>
</tr>
<tr>
<td>16.4.4</td>
<td>basic_ios iostate flags functions...........................................</td>
<td>422</td>
</tr>
<tr>
<td>16.4.4.1</td>
<td>operator bool.........................................................................</td>
<td>422</td>
</tr>
<tr>
<td>16.4.4.2</td>
<td>operator !.............................................................................</td>
<td>422</td>
</tr>
<tr>
<td>16.4.4.3</td>
<td>rdstate...............................................................................</td>
<td>423</td>
</tr>
<tr>
<td>16.4.4.4</td>
<td>clear....................................................................................</td>
<td>425</td>
</tr>
<tr>
<td>16.4.4.5</td>
<td>setstate...............................................................................</td>
<td>426</td>
</tr>
<tr>
<td>16.4.4.6</td>
<td>good.....................................................................................</td>
<td>427</td>
</tr>
<tr>
<td>16.4.4.7</td>
<td>eof.......................................................................................</td>
<td>427</td>
</tr>
<tr>
<td>16.4.4.8</td>
<td>fail......................................................................................</td>
<td>428</td>
</tr>
<tr>
<td>16.4.4.9</td>
<td>bad......................................................................................</td>
<td>429</td>
</tr>
<tr>
<td>16.4.4.10</td>
<td>exceptions..........................................................................</td>
<td>431</td>
</tr>
<tr>
<td>16.5</td>
<td>ios_base manipulators..................................................................</td>
<td>431</td>
</tr>
<tr>
<td>16.5.1</td>
<td>fmtflags manipulators.................................................................</td>
<td>432</td>
</tr>
<tr>
<td>16.5.2</td>
<td>adjustfield manipulators.......................................................</td>
<td>432</td>
</tr>
<tr>
<td>16.5.3</td>
<td>basefield manipulators...........................................................</td>
<td>433</td>
</tr>
<tr>
<td>16.5.4</td>
<td>floatfield manipulators............................................................</td>
<td>433</td>
</tr>
<tr>
<td>16.5.5</td>
<td>Overloading Manipulators.........................................................</td>
<td>434</td>
</tr>
</tbody>
</table>

**Chapter 17**

**Stream Buffers**

17.1 Stream buffer requirements.................................................................| 437  |

17.2 Class basic_streambuf.......................................................................| 438  |
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.2.1</td>
<td>basic_streambuf Constructor</td>
<td>438</td>
</tr>
<tr>
<td>17.2.1.1</td>
<td>Destructor</td>
<td>439</td>
</tr>
<tr>
<td>17.2.2</td>
<td>basic_streambuf Public Member Functions</td>
<td>439</td>
</tr>
<tr>
<td>17.2.2.1</td>
<td>Locales</td>
<td>439</td>
</tr>
<tr>
<td>17.2.2.2</td>
<td>basic_streambuf::pubimbue</td>
<td>439</td>
</tr>
<tr>
<td>17.2.2.3</td>
<td>basic_streambuf::getloc</td>
<td>440</td>
</tr>
<tr>
<td>17.2.3</td>
<td>Buffer Management and Positioning</td>
<td>440</td>
</tr>
<tr>
<td>17.2.3.1</td>
<td>basic_streambuf::pubsetbuf</td>
<td>440</td>
</tr>
<tr>
<td>17.2.3.2</td>
<td>basic_streambuf::pubseekoff</td>
<td>441</td>
</tr>
<tr>
<td>17.2.3.3</td>
<td>basic_streambuf::pubseekpos</td>
<td>442</td>
</tr>
<tr>
<td>17.2.3.4</td>
<td>basic_streambuf::pubsync</td>
<td>443</td>
</tr>
<tr>
<td>17.2.4</td>
<td>Get Area</td>
<td>444</td>
</tr>
<tr>
<td>17.2.4.1</td>
<td>basic_streambuf::in_avail</td>
<td>444</td>
</tr>
<tr>
<td>17.2.4.2</td>
<td>basic_streambuf::snextc</td>
<td>444</td>
</tr>
<tr>
<td>17.2.4.3</td>
<td>basic_streambuf::sbumpe</td>
<td>445</td>
</tr>
<tr>
<td>17.2.4.4</td>
<td>basic_streambuf::sgetc</td>
<td>446</td>
</tr>
<tr>
<td>17.2.4.5</td>
<td>basic_streambuf::sgetn</td>
<td>447</td>
</tr>
<tr>
<td>17.2.5</td>
<td>Putback</td>
<td>447</td>
</tr>
<tr>
<td>17.2.5.1</td>
<td>basic_streambuf::sputbackc</td>
<td>447</td>
</tr>
<tr>
<td>17.2.5.2</td>
<td>basic_streambuf::sungetc</td>
<td>449</td>
</tr>
<tr>
<td>17.2.6</td>
<td>Put Area</td>
<td>449</td>
</tr>
<tr>
<td>17.2.6.1</td>
<td>basic_streambuf::sputc</td>
<td>449</td>
</tr>
<tr>
<td>17.2.6.2</td>
<td>basic_streambuf::sputn</td>
<td>450</td>
</tr>
<tr>
<td>17.2.6.3</td>
<td>basic_streambuf Protected Member Functions</td>
<td>450</td>
</tr>
<tr>
<td>17.2.7</td>
<td>Get Area Access</td>
<td>451</td>
</tr>
<tr>
<td>17.2.7.1</td>
<td>basic_streambuf::eback</td>
<td>451</td>
</tr>
<tr>
<td>17.2.7.2</td>
<td>basic_streambuf::gptr</td>
<td>451</td>
</tr>
<tr>
<td>17.2.7.3</td>
<td>basic_streambuf::egptr</td>
<td>451</td>
</tr>
<tr>
<td>17.2.7.4</td>
<td>basic_streambuf::gbump</td>
<td>452</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>17.2.7.5</td>
<td>basic_streambuf::setg</td>
<td>452</td>
</tr>
<tr>
<td>17.2.8</td>
<td>Put Area Access</td>
<td>452</td>
</tr>
<tr>
<td>17.2.8.1</td>
<td>basic_streambuf::pbase</td>
<td>452</td>
</tr>
<tr>
<td>17.2.8.2</td>
<td>basic_streambuf::pptr</td>
<td>453</td>
</tr>
<tr>
<td>17.2.8.3</td>
<td>basic_streambuf::epptr</td>
<td>453</td>
</tr>
<tr>
<td>17.2.8.4</td>
<td>basic_streambuf::pbump</td>
<td>453</td>
</tr>
<tr>
<td>17.2.8.5</td>
<td>basic_streambuf::setp</td>
<td>453</td>
</tr>
<tr>
<td>17.2.9</td>
<td>basic_streambuf Virtual Functions</td>
<td>454</td>
</tr>
<tr>
<td>17.2.9.1</td>
<td>Locales</td>
<td>454</td>
</tr>
<tr>
<td>17.2.9.2</td>
<td>basic_streambuf::imbue</td>
<td>454</td>
</tr>
<tr>
<td>17.2.10</td>
<td>Buffer Management and Positioning</td>
<td>454</td>
</tr>
<tr>
<td>17.2.10.1</td>
<td>basic_streambuf::setbuf</td>
<td>454</td>
</tr>
<tr>
<td>17.2.10.2</td>
<td>basic_streambuf::seekoff</td>
<td>455</td>
</tr>
<tr>
<td>17.2.10.3</td>
<td>basic_streambuf::seekpos</td>
<td>455</td>
</tr>
<tr>
<td>17.2.10.4</td>
<td>basic_streambuf::sync</td>
<td>456</td>
</tr>
<tr>
<td>17.2.11</td>
<td>Get Area</td>
<td>456</td>
</tr>
<tr>
<td>17.2.11.1</td>
<td>basic_streambuf::showmanyc</td>
<td>456</td>
</tr>
<tr>
<td>17.2.11.2</td>
<td>basic_streambuf::xsgetn</td>
<td>456</td>
</tr>
<tr>
<td>17.2.11.3</td>
<td>basic_streambuf::underflow</td>
<td>457</td>
</tr>
<tr>
<td>17.2.11.4</td>
<td>basic_streambuf::uflow</td>
<td>457</td>
</tr>
<tr>
<td>17.2.12</td>
<td>Putback</td>
<td>458</td>
</tr>
<tr>
<td>17.2.12.1</td>
<td>basic_streambuf::pbackfail</td>
<td>458</td>
</tr>
<tr>
<td>17.2.13</td>
<td>Put Area</td>
<td>458</td>
</tr>
<tr>
<td>17.2.13.1</td>
<td>basic_streambuf::xsputn</td>
<td>458</td>
</tr>
<tr>
<td>17.2.13.2</td>
<td>basic_streambuf::overflow</td>
<td>459</td>
</tr>
</tbody>
</table>

Chapter 18  
Formatting and Manipulators

18.1 Headers.............................................................................................................. 461
18.2 Input Streams........................................................................................................ 461
## Template class basic_istream

18.2.1.1 basic_istream Constructors ................................................................. 462
18.2.1.2 Destructor ............................................................................................. 462

## Class basic_istream::sentry

18.2.2.1 Class basic_istream::sentry Constructor ........................................... 463
18.2.2.2 Destructor ............................................................................................. 464
18.2.2.3 sentry::Operator bool ............................................................................ 464

## Formatted input functions

18.2.3.1 Common requirements .......................................................................... 464
18.2.3.2 Arithmetic Extractors Operator >> ......................................................... 464
18.2.3.3 basic_istream extractor operator >> ..................................................... 465
18.2.3.4 Overloading Extractors ......................................................................... 467

## Unformatted input functions

18.2.4.1 basic_istream::gcount ........................................................................ 470
18.2.4.2 basic_istream::get ............................................................................... 471
18.2.4.2.1 Remarks ........................................................................................... 472
18.2.4.3 basic_istream::getline ........................................................................ 473
18.2.4.4 basic_istream::ignore ......................................................................... 475
18.2.4.5 basic_istream::peek ............................................................................ 476
18.2.4.6 basic_istream::read ............................................................................. 477
18.2.4.7 basic_istream::readsome ................................................................. 478
18.2.4.8 basic_istream::putback ..................................................................... 479
18.2.4.9 basic_istream::unget ........................................................................... 481
18.2.4.10 basic_istream::sync ......................................................................... 482
18.2.4.11 basic_istream::tellg ......................................................................... 483
18.2.4.12 basic_istream::seekg ....................................................................... 483

## Standard basic_istream manipulators

18.2.5.1 basic_ifstream::ws ............................................................................ 485
18.2.5.2 basic_iostream Constructor ............................................................... 487
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2.5.2.1</td>
<td>Destructor</td>
<td>487</td>
</tr>
<tr>
<td>18.3</td>
<td>Output streams</td>
<td>487</td>
</tr>
<tr>
<td>18.3.1</td>
<td>Template class basic_ostream</td>
<td>487</td>
</tr>
<tr>
<td>18.3.1.1</td>
<td>basic_ostream Constructor</td>
<td>488</td>
</tr>
<tr>
<td>18.3.1.2</td>
<td>Destructor</td>
<td>488</td>
</tr>
<tr>
<td>18.3.2</td>
<td>Class basic_ostream::sentry</td>
<td>489</td>
</tr>
<tr>
<td>18.3.2.1</td>
<td>Class basic_ostream::sentry Constructor</td>
<td>489</td>
</tr>
<tr>
<td>18.3.2.2</td>
<td>Destructor</td>
<td>489</td>
</tr>
<tr>
<td>18.3.2.3</td>
<td>sentry::Operator bool</td>
<td>490</td>
</tr>
<tr>
<td>18.3.3</td>
<td>Formatted output functions</td>
<td>490</td>
</tr>
<tr>
<td>18.3.3.1</td>
<td>Common requirements</td>
<td>490</td>
</tr>
<tr>
<td>18.3.3.2</td>
<td>Arithmetic Inserter Operator &lt;&lt;</td>
<td>490</td>
</tr>
<tr>
<td>18.3.3.3</td>
<td>basic_ostream::operator&lt;&lt;</td>
<td>492</td>
</tr>
<tr>
<td>18.3.3.4</td>
<td>Overloading Inserters</td>
<td>493</td>
</tr>
<tr>
<td>18.3.4</td>
<td>Unformatted output functions</td>
<td>495</td>
</tr>
<tr>
<td>18.3.4.1</td>
<td>basic_ostream::tellp</td>
<td>495</td>
</tr>
<tr>
<td>18.3.4.2</td>
<td>basic_ostream::seekp</td>
<td>495</td>
</tr>
<tr>
<td>18.3.4.3</td>
<td>basic_ostream::put</td>
<td>497</td>
</tr>
<tr>
<td>18.3.4.4</td>
<td>basic_ostream::write</td>
<td>497</td>
</tr>
<tr>
<td>18.3.4.5</td>
<td>basic_ostream::flush</td>
<td>499</td>
</tr>
<tr>
<td>18.3.5</td>
<td>Standard basic_ostream manipulators</td>
<td>501</td>
</tr>
<tr>
<td>18.3.5.1</td>
<td>basic_ostream::endl</td>
<td>501</td>
</tr>
<tr>
<td>18.3.5.2</td>
<td>basic_ostream::ends</td>
<td>502</td>
</tr>
<tr>
<td>18.3.5.3</td>
<td>basic_ostream::flush</td>
<td>503</td>
</tr>
<tr>
<td>18.4</td>
<td>Standard manipulators</td>
<td>504</td>
</tr>
<tr>
<td>18.4.1</td>
<td>Standard Manipulator Instantiations</td>
<td>504</td>
</tr>
<tr>
<td>18.4.2</td>
<td>resetiosflags</td>
<td>505</td>
</tr>
<tr>
<td>18.4.3</td>
<td>setiosflags</td>
<td>505</td>
</tr>
<tr>
<td>18.4.4</td>
<td>setbase</td>
<td>506</td>
</tr>
</tbody>
</table>
### Section 18.4.5
- **setfill**
  - Page 507

### Section 18.4.6
- **setprecision**
  - Page 507

### Section 18.4.7
- **setw**
  - Page 508

### Section 18.4.8
- **Overloaded Manipulator**
  - Page 509

---

**Chapter 19**

**String Based Streams**

### Section 19.1
- **Header <sstream>**
  - Page 511

### Section 19.2
- **Template class basic_stringbuf**
  - Page 511
  - **basic_stringbuf constructors**
    - Page 512
  - **Member functions**
    - Page 513
    - **basic_stringbuf::str**
      - Page 513
  - **Overridden virtual functions**
    - Page 514
    - **basic_stringbuf::underflow**
      - Page 514
    - **basic_stringbuf::pbackfail**
      - Page 514
    - **basic_stringbuf::overflow**
      - Page 515
    - **basic_stringbuf::seekoff**
      - Page 515
    - **basic_stringbuf::seekpos**
      - Page 516

### Section 19.3
- **Template class basic_istringstream**
  - Page 516
  - **basic_istringstream Constructor**
    - Page 516

### Section 19.4
- **Class basic_ostringstream**
  - Page 519
  - **basic_ostringstream Constructor**
    - Page 519

### Section 19.5
- **Class basic_stringstream**
  - Page 522
  - **basic_stringstream Constructor**
    - Page 523
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.5.2</td>
<td>Member functions</td>
<td>524</td>
</tr>
<tr>
<td>19.5.2.1</td>
<td>basic_stringstream::rdbuf</td>
<td>524</td>
</tr>
<tr>
<td>19.5.2.2</td>
<td>basic_stringstream::str</td>
<td>525</td>
</tr>
<tr>
<td>20.1</td>
<td>Header fstream</td>
<td>527</td>
</tr>
<tr>
<td>20.2</td>
<td>File Streams Type Defines</td>
<td>527</td>
</tr>
<tr>
<td>20.3</td>
<td>Template class basic_filebuf</td>
<td>528</td>
</tr>
<tr>
<td>20.3.1</td>
<td>basic_filebuf Constructors</td>
<td>528</td>
</tr>
<tr>
<td>20.3.1.1</td>
<td>Constructor</td>
<td>528</td>
</tr>
<tr>
<td>20.3.1.2</td>
<td>Destructor</td>
<td>529</td>
</tr>
<tr>
<td>20.3.2</td>
<td>Member functions</td>
<td>529</td>
</tr>
<tr>
<td>20.3.2.1</td>
<td>basic_filebuf::is_open</td>
<td>529</td>
</tr>
<tr>
<td>20.3.2.2</td>
<td>basic_filebuf::open</td>
<td>529</td>
</tr>
<tr>
<td>20.3.2.3</td>
<td>basic_filebuf::close</td>
<td>531</td>
</tr>
<tr>
<td>20.3.3</td>
<td>Overridden virtual functions</td>
<td>531</td>
</tr>
<tr>
<td>20.3.3.1</td>
<td>basic_filebuf::showmanyc</td>
<td>531</td>
</tr>
<tr>
<td>20.3.3.2</td>
<td>basic_filebuf::underflow</td>
<td>532</td>
</tr>
<tr>
<td>20.3.3.3</td>
<td>basic_filebuf::pbackfail</td>
<td>532</td>
</tr>
<tr>
<td>20.3.3.4</td>
<td>basic_filebuf::overflow</td>
<td>532</td>
</tr>
<tr>
<td>20.3.3.5</td>
<td>basic_filebuf::seekoff</td>
<td>533</td>
</tr>
<tr>
<td>20.3.3.6</td>
<td>basic_filebuf::seekpos</td>
<td>533</td>
</tr>
<tr>
<td>20.3.3.7</td>
<td>basic_filebuf::setbuf</td>
<td>533</td>
</tr>
<tr>
<td>20.3.3.8</td>
<td>basic_filebuf::sync</td>
<td>534</td>
</tr>
<tr>
<td>20.3.3.9</td>
<td>basic_filebuf::imbue</td>
<td>534</td>
</tr>
<tr>
<td>20.4</td>
<td>Template class basic_ifstream</td>
<td>534</td>
</tr>
<tr>
<td>20.4.1</td>
<td>basic_ifstream Constructor</td>
<td>534</td>
</tr>
<tr>
<td>20.4.2</td>
<td>Member functions</td>
<td>535</td>
</tr>
<tr>
<td>20.4.2.1</td>
<td>basic_ifstream::rdbuf</td>
<td>536</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>20.4.2.2</td>
<td>basic_ifstream::is_open</td>
<td>536</td>
</tr>
<tr>
<td>20.4.2.3</td>
<td>basic_ifstream::open</td>
<td>537</td>
</tr>
<tr>
<td>20.4.2.4</td>
<td>basic_ifstream::close</td>
<td>538</td>
</tr>
<tr>
<td>20.5</td>
<td>Template class basic_ofstream</td>
<td>538</td>
</tr>
<tr>
<td>20.5.1</td>
<td>basic_ofstream Constructors</td>
<td>539</td>
</tr>
<tr>
<td>20.5.2</td>
<td>Member functions</td>
<td>540</td>
</tr>
<tr>
<td>20.5.2.1</td>
<td>basic_ofstream::rdbuf</td>
<td>540</td>
</tr>
<tr>
<td>20.5.2.2</td>
<td>basic_ofstream::is_open</td>
<td>541</td>
</tr>
<tr>
<td>20.5.2.3</td>
<td>basic_ofstream::open</td>
<td>541</td>
</tr>
<tr>
<td>20.5.2.4</td>
<td>basic_ofstream::close</td>
<td>543</td>
</tr>
<tr>
<td>20.6</td>
<td>Template class basic_fstream</td>
<td>543</td>
</tr>
<tr>
<td>20.6.1</td>
<td>basic_fstream Constructor</td>
<td>543</td>
</tr>
<tr>
<td>20.6.2</td>
<td>Member Functions</td>
<td>544</td>
</tr>
<tr>
<td>20.6.2.1</td>
<td>basic_fstream::rdbuf</td>
<td>544</td>
</tr>
<tr>
<td>20.6.2.2</td>
<td>basic_fstream::is_open</td>
<td>545</td>
</tr>
<tr>
<td>20.6.2.3</td>
<td>basic_fstream::open</td>
<td>546</td>
</tr>
<tr>
<td>20.6.2.4</td>
<td>basic_fstream::close</td>
<td>547</td>
</tr>
</tbody>
</table>

Chapter 21
C Library Files

Chapter 22
Strstream

22.1 Header strstream | 551
22.2 Strstreambuf Class | 551
22.2.1 Strstreambuf constructors and Destructors | 552
22.2.1.1 Constructors | 552
22.2.1.2 Destructor | 553
22.2.2 Strstreambuf Public Member Functions | 553
22.2.2.1 freeze | 553
22.2.2.2 pcount | 554
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.2.2.3</td>
<td>str</td>
<td>554</td>
</tr>
<tr>
<td>22.2.3</td>
<td>Protected Virtual Member Functions</td>
<td>555</td>
</tr>
<tr>
<td>22.2.3.1</td>
<td>setbuf</td>
<td>555</td>
</tr>
<tr>
<td>22.2.3.2</td>
<td>seekoff</td>
<td>555</td>
</tr>
<tr>
<td>22.2.3.3</td>
<td>seekpos</td>
<td>556</td>
</tr>
<tr>
<td>22.2.3.4</td>
<td>underflow</td>
<td>556</td>
</tr>
<tr>
<td>22.2.3.5</td>
<td>pbackfail</td>
<td>557</td>
</tr>
<tr>
<td>22.2.3.6</td>
<td>overflow</td>
<td>557</td>
</tr>
<tr>
<td>22.3</td>
<td>iostream Class</td>
<td>558</td>
</tr>
<tr>
<td>22.3.1</td>
<td>Constructors and Destructor</td>
<td>558</td>
</tr>
<tr>
<td>22.3.1.1</td>
<td>Constructors</td>
<td>558</td>
</tr>
<tr>
<td>22.3.1.2</td>
<td>Destructor</td>
<td>559</td>
</tr>
<tr>
<td>22.3.2</td>
<td>Public Member Functions</td>
<td>559</td>
</tr>
<tr>
<td>22.3.2.1</td>
<td>rdbuf</td>
<td>559</td>
</tr>
<tr>
<td>22.3.2.2</td>
<td>str</td>
<td>560</td>
</tr>
<tr>
<td>22.4</td>
<td>ostream Class</td>
<td>560</td>
</tr>
<tr>
<td>22.4.1</td>
<td>Constructors and Destructor</td>
<td>560</td>
</tr>
<tr>
<td>22.4.1.1</td>
<td>Constructors</td>
<td>560</td>
</tr>
<tr>
<td>22.4.1.2</td>
<td>Destructor</td>
<td>561</td>
</tr>
<tr>
<td>22.4.2</td>
<td>Public Member Functions</td>
<td>561</td>
</tr>
<tr>
<td>22.4.2.1</td>
<td>freeze</td>
<td>561</td>
</tr>
<tr>
<td>22.4.2.2</td>
<td>pcount</td>
<td>562</td>
</tr>
<tr>
<td>22.4.2.3</td>
<td>rdbuf</td>
<td>563</td>
</tr>
<tr>
<td>22.4.2.4</td>
<td>str</td>
<td>563</td>
</tr>
<tr>
<td>22.5</td>
<td>stringstream Class</td>
<td>564</td>
</tr>
<tr>
<td>22.5.1</td>
<td>stringstream Types</td>
<td>564</td>
</tr>
<tr>
<td>22.5.2</td>
<td>Constructors and Destructor</td>
<td>564</td>
</tr>
<tr>
<td>22.5.2.1</td>
<td>Constructors</td>
<td>564</td>
</tr>
<tr>
<td>22.5.2.2</td>
<td>Destructor</td>
<td>564</td>
</tr>
</tbody>
</table>
Chapter 23
Bitvector Class Library

23.1 Nested types

23.1.1 allocator_type

23.1.2 size_type

23.1.3 difference_type

23.1.4 value_type

23.1.5 reference

23.1.6 const_reference

23.1.7 iterators and pointers

23.2 Constructors

23.2.1 Destructor

23.2.2 Assignment

23.3 Capacity

23.3.1 size

23.3.2 empty

23.3.3 capacity

23.3.4 max_size

23.3.5 reserve

23.3.6 get_allocator

23.4 Iteration

23.5 Access

23.5.1 front

23.6 Insertion
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.6.1</td>
<td>push_back</td>
<td>577</td>
</tr>
<tr>
<td>23.6.2</td>
<td>insert</td>
<td>578</td>
</tr>
<tr>
<td>23.7</td>
<td>Erasure</td>
<td>579</td>
</tr>
<tr>
<td>23.7.1</td>
<td>pop_back</td>
<td>579</td>
</tr>
<tr>
<td>23.7.2</td>
<td>clear</td>
<td>579</td>
</tr>
<tr>
<td>23.7.3</td>
<td>erase</td>
<td>579</td>
</tr>
<tr>
<td>23.8</td>
<td>Miscellaneous</td>
<td>580</td>
</tr>
<tr>
<td>23.8.1</td>
<td>resize</td>
<td>580</td>
</tr>
<tr>
<td>23.8.2</td>
<td>swap</td>
<td>580</td>
</tr>
<tr>
<td>23.8.3</td>
<td>flip</td>
<td>581</td>
</tr>
<tr>
<td>23.8.4</td>
<td>invariants</td>
<td>581</td>
</tr>
<tr>
<td>23.9</td>
<td>Namespace scope functions</td>
<td>581</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 24</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EWL_Utility</strong></td>
<td></td>
</tr>
<tr>
<td>24.1</td>
<td>Header ewl_utility</td>
<td>583</td>
</tr>
<tr>
<td>24.2</td>
<td>Basic Compile-Time Transformations</td>
<td>584</td>
</tr>
<tr>
<td>24.2.1</td>
<td>remove_const</td>
<td>584</td>
</tr>
<tr>
<td>24.2.2</td>
<td>remove_volatile</td>
<td>584</td>
</tr>
<tr>
<td>24.2.3</td>
<td>remove_cv</td>
<td>585</td>
</tr>
<tr>
<td>24.2.4</td>
<td>remove_pointer</td>
<td>585</td>
</tr>
<tr>
<td>24.2.5</td>
<td>remove_reference</td>
<td>586</td>
</tr>
<tr>
<td>24.2.6</td>
<td>remove_bounds</td>
<td>586</td>
</tr>
<tr>
<td>24.2.7</td>
<td>remove_all</td>
<td>586</td>
</tr>
<tr>
<td>24.3</td>
<td>Type Query</td>
<td>587</td>
</tr>
<tr>
<td>24.3.1</td>
<td>is_same</td>
<td>587</td>
</tr>
<tr>
<td>24.4</td>
<td>CV Query</td>
<td>587</td>
</tr>
<tr>
<td>24.4.1</td>
<td>is_const</td>
<td>587</td>
</tr>
<tr>
<td>24.4.2</td>
<td>is_volatile</td>
<td>587</td>
</tr>
<tr>
<td>24.5</td>
<td>Type Classification</td>
<td>588</td>
</tr>
</tbody>
</table>
Chapter 25
EWL C++ Debug Mode

25.1 Overview of EWL C++ Debug Mode.................................................................................................................597
25.1.1 Types of Errors Detected..................................................................................................................................597
25.1.2 How to Enable Debug Mode............................................................................................................................597

25.2 Debug Mode Implementations..................................................................................................................................598
25.2.1 Debug Mode Containers........................................................................................................................................600
25.2.1.1 deque.........................................................................................................................................................600
25.2.1.2 list............................................................................................................................................................601
25.2.1.3 string........................................................................................................................................................601
25.2.1.4 vector........................................................................................................................................................602
25.2.1.5 tree-based containers - map, multimap, set, multiset..................................................................................603
25.2.1.6 cdeque.........................................................................................................................................................604
25.2.1.7 slist............................................................................................................................................................605
25.2.1.8 hash-based containers - map, multimap, set, multiset...............................................................................606
25.2.2 Invariants............................................................................................................................................................606

Chapter 26
Hash Libraries

26.1 General Hash Issues.................................................................................................................................................609
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.1.1</td>
<td>Introduction..................................................................................</td>
<td>609</td>
</tr>
<tr>
<td>26.1.2</td>
<td>Namespace Issues..........................................................................</td>
<td>610</td>
</tr>
<tr>
<td>26.1.2.1</td>
<td>Fully Qualified Reference................................................................</td>
<td>610</td>
</tr>
<tr>
<td>26.1.2.2</td>
<td>Namespace Alias..........................................................................</td>
<td>610</td>
</tr>
<tr>
<td>26.1.2.3</td>
<td>Using Declaration.......................................................................</td>
<td>611</td>
</tr>
<tr>
<td>26.1.2.4</td>
<td>Using Directive..........................................................................</td>
<td>611</td>
</tr>
<tr>
<td>26.1.2.5</td>
<td>Compatibility Headers..................................................................</td>
<td>612</td>
</tr>
<tr>
<td>26.1.2.6</td>
<td>Constructors.............................................................................</td>
<td>612</td>
</tr>
<tr>
<td>26.1.2.7</td>
<td>Iterator Issues...........................................................................</td>
<td>613</td>
</tr>
<tr>
<td>26.1.2.8</td>
<td>Capacity....................................................................................</td>
<td>613</td>
</tr>
<tr>
<td>26.1.2.9</td>
<td>insert.......................................................................................</td>
<td>615</td>
</tr>
<tr>
<td>26.1.2.10</td>
<td>insert.......................................................................................</td>
<td>616</td>
</tr>
<tr>
<td>26.1.2.11</td>
<td>erase.......................................................................................</td>
<td>616</td>
</tr>
<tr>
<td>26.1.2.12</td>
<td>Observers..................................................................................</td>
<td>617</td>
</tr>
<tr>
<td>26.1.2.13</td>
<td>Set Operations...........................................................................</td>
<td>617</td>
</tr>
<tr>
<td>26.1.2.14</td>
<td>Global Methods..........................................................................</td>
<td>618</td>
</tr>
<tr>
<td>26.1.3</td>
<td>Incompatibility with Previous versions of Hash Containers............</td>
<td>619</td>
</tr>
<tr>
<td>26.2</td>
<td>Hash_set....................................................................................</td>
<td>619</td>
</tr>
<tr>
<td>26.2.1</td>
<td>Introduction.............................................................................</td>
<td>619</td>
</tr>
<tr>
<td>26.2.2</td>
<td>Old Hashset Headers....................................................................</td>
<td>620</td>
</tr>
<tr>
<td>26.2.3</td>
<td>Template Parameters...................................................................</td>
<td>620</td>
</tr>
<tr>
<td>26.2.4</td>
<td>Nested Types.............................................................................</td>
<td>621</td>
</tr>
<tr>
<td>26.2.5</td>
<td>Iterator Issues..........................................................................</td>
<td>621</td>
</tr>
<tr>
<td>26.2.6</td>
<td>hash_set...................................................................................</td>
<td>621</td>
</tr>
<tr>
<td>26.3</td>
<td>Hash_map...................................................................................</td>
<td>621</td>
</tr>
<tr>
<td>26.3.1</td>
<td>Introduction.............................................................................</td>
<td>622</td>
</tr>
<tr>
<td>26.3.2</td>
<td>Old Hashmap Headers....................................................................</td>
<td>622</td>
</tr>
<tr>
<td>26.3.3</td>
<td>Template Parameters...................................................................</td>
<td>622</td>
</tr>
<tr>
<td>26.3.4</td>
<td>Nested Types.............................................................................</td>
<td>623</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>26.3.5</td>
<td>Iterator Issues</td>
<td>624</td>
</tr>
<tr>
<td>26.3.6</td>
<td>Element Access</td>
<td>624</td>
</tr>
<tr>
<td>26.4</td>
<td>Hash_fun</td>
<td>624</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 27</strong></td>
<td></td>
</tr>
<tr>
<td>27.1</td>
<td>Overview of EWL Threads</td>
<td>627</td>
</tr>
<tr>
<td>27.2</td>
<td>Mutex and Locks</td>
<td>628</td>
</tr>
<tr>
<td>27.3</td>
<td>Threads</td>
<td>631</td>
</tr>
<tr>
<td>27.4</td>
<td>Condition Variables</td>
<td>633</td>
</tr>
<tr>
<td>27.5</td>
<td>call_once</td>
<td>638</td>
</tr>
<tr>
<td>27.6</td>
<td>thread_specific_ptr</td>
<td>640</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 28</strong></td>
<td></td>
</tr>
<tr>
<td>28.1</td>
<td>Overview of EWL Implementation of Technical Report 1</td>
<td>643</td>
</tr>
<tr>
<td>28.2</td>
<td>Template class Sig class result_of</td>
<td>643</td>
</tr>
<tr>
<td>28.2.1</td>
<td>result_of</td>
<td>644</td>
</tr>
<tr>
<td>28.2.2</td>
<td>Public Members</td>
<td>644</td>
</tr>
<tr>
<td>28.2.2.1</td>
<td>get_result_type</td>
<td>644</td>
</tr>
<tr>
<td>28.3</td>
<td>Template class T class reference_wrapper</td>
<td>645</td>
</tr>
<tr>
<td>28.3.1</td>
<td>reference_wrapper</td>
<td>645</td>
</tr>
<tr>
<td>28.3.2</td>
<td>Public Member Functions</td>
<td>646</td>
</tr>
<tr>
<td>28.3.2.1</td>
<td>ref</td>
<td>646</td>
</tr>
<tr>
<td>28.3.2.2</td>
<td>cref</td>
<td>646</td>
</tr>
<tr>
<td>28.4</td>
<td>Template class Sig class function</td>
<td>646</td>
</tr>
<tr>
<td>28.4.1</td>
<td>Constructors Destructors and Assignment Operator</td>
<td>647</td>
</tr>
<tr>
<td>28.4.1.1</td>
<td>Constructor</td>
<td>647</td>
</tr>
<tr>
<td>28.4.1.2</td>
<td>Destructor</td>
<td>647</td>
</tr>
<tr>
<td>28.4.2</td>
<td>Public Member Functions</td>
<td>649</td>
</tr>
<tr>
<td>28.4.2.1</td>
<td>Member_function</td>
<td>649</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>28.5</td>
<td>Template class T class shared_ptr</td>
<td>649</td>
</tr>
<tr>
<td>28.6</td>
<td>Template class T class enable_shared_from_this</td>
<td>649</td>
</tr>
<tr>
<td>28.6.1</td>
<td>Constructors Destructors and Assignment Operator</td>
<td>650</td>
</tr>
<tr>
<td>28.6.1.1</td>
<td>Constructor</td>
<td>650</td>
</tr>
<tr>
<td>28.6.1.2</td>
<td>Destructor</td>
<td>650</td>
</tr>
<tr>
<td>28.6.2</td>
<td>Public Member Functions</td>
<td>651</td>
</tr>
<tr>
<td>28.6.2.1</td>
<td>Member_function</td>
<td>651</td>
</tr>
<tr>
<td>28.7</td>
<td>Template class T0, class T1, ... class T9 class tuple</td>
<td>651</td>
</tr>
<tr>
<td>28.7.1</td>
<td>Constructors Destructors and Assignment Operator</td>
<td>652</td>
</tr>
<tr>
<td>28.7.1.1</td>
<td>Constructor</td>
<td>652</td>
</tr>
<tr>
<td>28.7.1.2</td>
<td>Destructor</td>
<td>653</td>
</tr>
<tr>
<td>28.7.2</td>
<td>Public Member Functions</td>
<td>654</td>
</tr>
<tr>
<td>28.7.2.1</td>
<td>Member_function</td>
<td>654</td>
</tr>
<tr>
<td>28.8</td>
<td>Template bind</td>
<td>655</td>
</tr>
<tr>
<td>28.8.1</td>
<td>sort predicate</td>
<td>660</td>
</tr>
<tr>
<td>28.8.2</td>
<td>remove_if predicate</td>
<td>661</td>
</tr>
<tr>
<td>28.8.3</td>
<td>function</td>
<td>662</td>
</tr>
</tbody>
</table>

Chapter 29
Ewlconfig

29.1  C++ Switches, Flags and Defines........................................................................... 665

29.1.1  _CSTD................................................................................................................... 666
29.1.2  _Inhibit_Container_Optimization........................................................................ 666
29.1.3  _Inhibit_Optimize_RB_bit.................................................................................. 666
29.1.4  _EWL_DEBUG.......................................................................................................... 667
29.1.5  _ewl_error............................................................................................................ 667
29.1.6  _EWL_ARRAY_AUTO_PTR......................................................................................... 667
29.1.7  _EWL_CFILE_STREAM............................................................................................ 667
29.1.8  __EWL_CPP__........................................................................................................ 668
29.1.9  _EWL_EXTENDED_BINDERS..................................................................................... 668
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.1.10</td>
<td>_EWL_EXTENDED_PRECISION_OUTP</td>
<td>669</td>
</tr>
<tr>
<td>29.1.11</td>
<td>_EWL_FORCE_ENABLE_BOOL_SUPPORT</td>
<td>669</td>
</tr>
<tr>
<td>29.1.12</td>
<td>_EWL_FORCE_ENUMS_ALWAYS_INT</td>
<td>670</td>
</tr>
<tr>
<td>29.1.13</td>
<td>_EWL_IMP_EXP</td>
<td>670</td>
</tr>
<tr>
<td>29.1.14</td>
<td><strong>EWL_LONGLONG_SUPPORT</strong></td>
<td>671</td>
</tr>
<tr>
<td>29.1.15</td>
<td>_EWL_MINIMUM_NAMED_LOCALE</td>
<td>671</td>
</tr>
<tr>
<td>29.1.16</td>
<td>_EWL_NO_BOOL</td>
<td>672</td>
</tr>
<tr>
<td>29.1.17</td>
<td>_EWL_NO_CONSOLE_IO</td>
<td>672</td>
</tr>
<tr>
<td>29.1.18</td>
<td>_EWL_NO_CPP_NAMESPACE</td>
<td>672</td>
</tr>
<tr>
<td>29.1.19</td>
<td>_EWL_NO_EXCEPTIONS</td>
<td>672</td>
</tr>
<tr>
<td>29.1.20</td>
<td>_EWL_NO_EXPLICIT_FUNC_TEMPLATE_ARG</td>
<td>673</td>
</tr>
<tr>
<td>29.1.21</td>
<td>_EWL_NO_FILE_IO</td>
<td>674</td>
</tr>
<tr>
<td>29.1.22</td>
<td>_EWL_NO_IO</td>
<td>674</td>
</tr>
<tr>
<td>29.1.23</td>
<td>_EWL_NO_LOCALE</td>
<td>674</td>
</tr>
<tr>
<td>29.1.24</td>
<td>_EWL_NO_REFCOUNT_STRING</td>
<td>674</td>
</tr>
<tr>
<td>29.1.25</td>
<td>_EWL_NO_VECTOR_BOOL</td>
<td>674</td>
</tr>
<tr>
<td>29.1.26</td>
<td>_EWL_NO_WCHART</td>
<td>675</td>
</tr>
<tr>
<td>29.1.27</td>
<td>_EWL_NO_WCHART_LANG_SUPPORT</td>
<td>675</td>
</tr>
<tr>
<td>29.1.28</td>
<td>_EWL_NO_WCHART_C_SUPPORT</td>
<td>675</td>
</tr>
<tr>
<td>29.1.29</td>
<td>_EWL_NO_WCHART_CPP_SUPPORT</td>
<td>675</td>
</tr>
<tr>
<td>29.1.30</td>
<td>_EWL_POSIX_STREAM</td>
<td>676</td>
</tr>
<tr>
<td>29.1.31</td>
<td>_EWL_WIDE_FILENAME</td>
<td>676</td>
</tr>
<tr>
<td>29.1.32</td>
<td>_EWL_WFILEIO_AVAILABLE</td>
<td>676</td>
</tr>
<tr>
<td>29.1.33</td>
<td>_STD</td>
<td>677</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

This reference manual describes the contents of the Embedded Warrior Library for C++. The C++ Standard library provides an extensible framework, and contains components for: language support, diagnostics, general utilities, strings, locales, containers, iterators, algorithms, numerics, and input/output. Additionally, EWL C++ offers extra facilities for input/output, threads, and other components.

1.1 About the EWL C++ Library Reference Manual

This section describes each chapter in this manual. The various chapter's layout is guided by the ISO (International Organization for Standardization) C++ Standard.

The EWL C++ Library Overview of this manual describes the language support library that provides components that are required by certain parts of the C++ language, such as memory allocation and exception processing.

Language Support Library discusses the ANSI/ISO language support library.

Diagnostics Library elaborates on the diagnostics library that provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.

General Utilities Libraries discusses the general utilities library, which includes components used by other library elements, such as predefined storage allocator for dynamic storage management.

Strings Library discusses the strings components provided for manipulating text represented as sequences of type char, sequences of type wchar_t, or sequences of any other "character-like" type.

Localization Library covers the localization components extend internationalization support for character classification, numeric, monetary, and date/time formatting and parsing among other things.
Containers Library discusses container classes: lists, vectors, stacks, and so forth. These classes provide a C++ program with access to a subset of the most widely used algorithms and data structures.

Iterators Library discusses iterator classes.

Algorithms Library discusses the algorithms library. This library provides sequence, sorting, and general numerics algorithms.

The Numerics Library (clause 26) discusses the numerics library. It describes numeric arrays, generalized numeric algorithms and facilities included from the ISO C library.

Complex Class describes the components for complex number types

Input and Output Library overviews the input and output class libraries.

The Streams and String Forward Declarations discusses the input and output streams forward declarations.

The Standard Input and Output Stream Library discusses the initialized input and output objects.

Iostreams Base Classes discusses the iostream_base class.

Stream Buffers discusses the stream buffer classes.

Formatting and Manipulators discusses the formatting and manipulator classes.

String Based Streams discusses the string based stream classes.

File Based Streams discusses the file based stream classes.

C Library Files discusses the namespace C Library functions.

The Strstream Class Library (Annex D) discusses the non standard string stream classes.

Bitvector Class Library discusses the boolean vector class library.

EWL Utility utilities used for non standard headers.

Overview of EWL C++ Debug Mode describes the Embedded Warrior Library for C++ debug mode facilities.

Hash Libraries describes nonstandard "hash" libraries.

Metrowerks::threads is a reference to threads support in the Embedded Warrior Libraries.

EWL std::tr1 is a reference about items that are proposed for inclusion in the Embedded Warrior Library.
C++ Switches, Flags and Defines is a chapter on the various flags that you can use to create a customized version of the EWL C++ Library.
Chapter 2
The C++ Library

This chapter is an introduction to the Embedded Warrior Library for C++.

2.1 The EWL C++ Library Overview

This section introduces you to the definitions, conventions, terminology, and other aspects of the EWL C++ library.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Definitions standard C++ terminology
- Additional Definitions additional terminology
- Multi-Thread Safety multi-threaded policy
- Methods of Descriptions standard conventions
- Library-wide Requirements library requirements

2.2 Definitions

This section discusses the meaning of certain terms in the EWL C++ library.

- Arbitrary-Positional Stream
- Character
- Character Sequences
- Comparison Function
- Component
- Default Behavior
2.2.1 Arbitrary-Positional Stream

A stream that can seek to any position within the length of the stream. An arbitrary-positional stream is also a repositional stream.

2.2.2 Character

Any object which, when treated sequentially, can represent text. A character can be represented by any type that provides the definitions specified.

2.2.3 Character Sequences

A class or a type used to represent a character. A character container class shall be a POD type.

2.2.4 Comparison Function

An operator function for equality or relational operators.
2.2.5 Component

A group of library entities directly related as members, parameters, or return types. For example, a class and a related non-member template function entity would referred to as a component.

2.2.6 Default Behavior

The specific behavior provided by the implementation, for replacement and handler functions.

2.2.7 Handler Function

A non-reserved function that may be called at various points with a program through supplying a pointer to the function. The definition may be provided by a C++ program.

2.2.8 Iostream Class Templates

Templates that take two template arguments: charT and traits. CharT is a character container class, and traits is a structure which defines additional characteristics and functions of the character type.

2.2.9 Modifier Function

A class member function other than constructors, assignment, or destructor, that alters the state of an object of the class.

2.2.10 Object State

The current value of all non-static class members of an object.
2.2.11 Narrow-oriented iostream Classes

The instantiations of the iostream class templates on the character container class. Traditional iostream classes are regarded as the narrow-oriented iostream classes.

2.2.12 NTCTS

Null Terminated Character Type Sequences. Traditional char strings are NTCTS.

2.2.13 Observer Function

A const member function that accesses the state of an object of the class, but does not alter that state.

2.2.14 Replacement Function

A non-reserved C++ function whose definition is provided by a program. Only one definition for such a function is in effect for the duration of the program's execution.

2.2.15 Required Behavior

The behavior for any replacement or handler function definition in the program replacement or handler function. If a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined.

2.2.16 Repositional Stream

A stream that can seek only to a position that was previously encountered.
2.2.17 Reserved Function

A function, specified as part of the C++ Standard Library, that must be defined by the implementation. If a C++ program provides a definition for any reserved function, the results are undefined.

2.2.18 Traits

A class that encapsulates a set of types and functions necessary for template classes and template functions to manipulate objects of types for which they are instantiated.

2.2.19 Wide-oriented IOSTREAM Classes

The instantiations of the `iostream` class templates on the character container class `wchar_t` and the default value of the traits parameter.

2.3 Additional Definitions


2.3.1 Multi-Thread Safety

EWL C++ Library is multi-thread safe provided that the operating system supports thread-safe system calls.

Library has locks at appropriate places in the code for thread safety. The locks are implemented as a mutex class -- the implementation of which may differ from platform to platform.
This ensures that the library is MT-Safe internally. For example, if a buffer is shared between two string class objects (via an internal refcount), then only one string object will be able to modify the shared buffer at a given time.

Thus the library will work in the presence of multiple threads in the same way as in single thread provided the user does not share objects between threads or locks between accesses to objects that are shared.

### 2.3.1.1 EWL C++ Thread Safety Policy

EWL C++ is Level-1 thread safe. That is:

- It is safe to simultaneously call const and non-const methods from different threads to distinct objects.
- It is safe to simultaneously call const methods, and methods from different threads to the same object as long as such methods and const methods
  - Are guaranteed to not alter the state of an object
  - Do not invalidate outstanding references or iterators of a container
- It is not safe for different threads to simultaneously access the same object when at least one thread calls non-const methods, or methods that invalidate outstanding references or iterators to the object. The programmer is responsible for using thread synchronization primitives (e.g. mutex) to avoid such situations.

Simultaneous use of allocators such as new and malloc are thread safe.

Simultaneous use of global objects such as cin and cout is not safe. The programmer is responsible for using thread synchronization primitives to avoid such situations. EWL C++ provides an extension to standard C++ (std::mutex) to aid in such code. For example:

### Listing: EWL Mutex Example

```cpp
#include <iostream>
#include <iomanip>
#include <mutex.h>

std::mutex cout_lock;

int main()
{
    cout_lock.lock();
    std::cout << "The number is " << 
    std::setw(5) << 20 << '\n';
    cout_lock.unlock();
}
```
Note that if only one thread is accessing a standard stream then no synchronization is necessary. For example, one could have one thread handling input from `cin`, and another thread handling output to `cout`, without worrying about mutex objects.

### 2.4 Methods of Descriptions

Conventions used to describe the C++ Standard Library.

#### 2.4.1 Structure of each sub-clause

The Embedded Warrior Library descriptions include a short description, notes, remarks, cross-references, and examples of usage.

#### 2.4.2 Other Conventions

Some other terminology and conventions used in this reference.

#### 2.4.2.1 Character sequences

- A letter is any of the 26 lowercase or 26 uppercase letters
- The decimal-point character is represented by a period, `.`
- A character sequence is an array object of the types `char`, `unsigned char`, or `signed char`.
- A character sequence can be designated by a pointer value `S` that points to its first element.

#### 2.4.2.2 Byte strings

- A null-terminated byte string, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the terminating null character).
• The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.
• The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.
• A static NTBS is an NTBS with static storage duration.

2.4.2.3 Multibyte strings

• A null-terminated multibyte string, or NTMBS, is an NTBS that consists of multibyte characters.
• A static NTMBS is an NTMBS with static storage duration.

2.4.2.4 Wide-character sequences

• A wide-character sequence is an array object of type wchar_t
• A wide character sequence can be designated by a pointer value that designates its first element.
• A null-terminated wide-character string, or NTWCS, is a wide-character sequence whose highest addressed element has the value zero.
• The length of an NTWCS is the number of elements that precede the terminating null wide character.
• An empty NTWCS has a length of zero.
• The value of an NTWCS is the sequence of values of the elements up to and including the terminating null character.
• A static NTWCS is an NTWCS with static storage duration.

2.4.2.5 Functions within classes

Some procedures, copy constructors, assignment operators, (non-virtual) destructors or virtual destructors, that can be generated by default may not be described.

2.4.2.6 Private members

To simplify understanding, where objects of certain types are required by the external specifications of their classes to store data. The declarations for such member objects are enclosed in a comment that ends with exposition only, as in:
2.5 Library-wide Requirements

The requirements that apply to the entire C++ Standard library.

- Library contents and organization
- Using the library
- Constraints on programs
- Conforming Implementations
- Reentrancy

2.5.1 Library contents and organization

The Embedded Warrior Library is organized in the same fashion as the ANSI/ISO C++ Standard.

2.5.1.1 Library Contents

Definitions are provided for Macros, Values, Types, Templates, Classes, Function and, Objects.

All library entities except macros, operator new and operator delete are defined within the namespace std or namespace nested within namespace std.

2.5.1.2 Headers

The components of the EWL C++ Library are declared or defined in various headers.

Table 2-1. EWL C++ Library headers:

<table>
<thead>
<tr>
<th>C++</th>
<th>Headers</th>
<th>C++</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;algorithm&gt;</td>
<td>&lt;bitset&gt;</td>
<td>&lt;complex&gt;</td>
<td>&lt;deque&gt;</td>
</tr>
<tr>
<td>&lt;exception&gt;</td>
<td>&lt;fstream&gt;</td>
<td>&lt;functional&gt;</td>
<td>&lt;iomanip&gt;</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Table 2-1. EWL C++ Library headers: (continued)

<table>
<thead>
<tr>
<th>C++ Headers</th>
<th>C++ Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;ios&gt;</code></td>
<td><code>&lt;iosfwd&gt;</code></td>
</tr>
<tr>
<td><code>&lt;iterator&gt;</code></td>
<td><code>&lt;limits&gt;</code></td>
</tr>
<tr>
<td><code>&lt;map&gt;</code></td>
<td><code>&lt;memory&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ostream&gt;</code></td>
<td><code>&lt;list&gt;</code></td>
</tr>
<tr>
<td><code>&lt;stack&gt;</code></td>
<td><code>&lt;string&gt;</code></td>
</tr>
<tr>
<td><code>&lt;stdio&gt;</code></td>
<td><code>&lt;iostream&gt;</code></td>
</tr>
<tr>
<td><code>&lt;stdlib&gt;</code></td>
<td><code>&lt;istream&gt;</code></td>
</tr>
<tr>
<td><code>&lt;stdlib&gt;</code></td>
<td><code>&lt;locale&gt;</code></td>
</tr>
<tr>
<td><code>&lt;stdlist&gt;</code></td>
<td><code>&lt;numeric&gt;</code></td>
</tr>
<tr>
<td><code>&lt;strbuff&gt;</code></td>
<td><code>&lt;set&gt;</code></td>
</tr>
<tr>
<td><code>&lt;string&gt;</code></td>
<td><code>&lt;sstream&gt;</code></td>
</tr>
<tr>
<td><code>&lt;temp&gt;</code></td>
<td><code>&lt;streambuf&gt;</code></td>
</tr>
<tr>
<td><code>&lt;typeinfo&gt;</code></td>
<td><code>&lt;valarray&gt;</code></td>
</tr>
<tr>
<td><code>&lt;utility&gt;</code></td>
<td><code>&lt;vector&gt;</code></td>
</tr>
</tbody>
</table>

### C Functional Headers

<table>
<thead>
<tr>
<th>C Functional Headers</th>
<th>C Functional Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;cassert&gt;</code></td>
<td><code>&lt;cerrno&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cctype&gt;</code></td>
<td><code>&lt;clocale&gt;</code></td>
</tr>
<tr>
<td><code>&lt;climits&gt;</code></td>
<td><code>&lt;cmath&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cfloat&gt;</code></td>
<td><code>&lt;ctime&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ciso646&gt;</code></td>
<td><code>&lt;cwchar&gt;</code></td>
</tr>
<tr>
<td><code>&lt;csignal&gt;</code></td>
<td><code>&lt;cwctype&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstdarg&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;cstddef&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;cstdio&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;cstdlib&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;cstring&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ctime&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

Unless noted otherwise, the contents of each C style header `cname` will be the same as that of the corresponding header `name.h`. In the EWL C++ Library the declarations and definitions (except for names which are defined as macros in C) are within namespace scope of the namespace std.

**NOTE**

The names defined as macros in C include: `assert`, `errno`, `offsetof`, `setjmp`, `va_arg`, `va_end`, and `va_start`.

### 2.5.1.3 Freestanding Implementations

A freestanding implementation has an implementation-defined set of headers. This set shall include at least the following headers.

<table>
<thead>
<tr>
<th>Table 2-2. EWL C++ Freestanding Implementation Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td><code>&lt;cstddef&gt;</code></td>
</tr>
<tr>
<td><code>&lt;limits&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
</tbody>
</table>
The Embedded Warrior Library header `<cstdlib>` includes the functions abort(), atexit(), and exit().

### 2.5.2 Using the library

A description of how a C++ program gains access to the facilities of the C++ Standard Library.

#### 2.5.2.1 Headers

A header's contents are made available to a translation unit when it contains the appropriate `#include` preprocessing directive.

A translation unit shall include a header only outside of any external declaration or definition, and shall include the header lexically before the first reference to any of the entities it declares or first defines in that translation unit.

#### 2.5.2.2 Linkage

The Embedded Warrior Library for C++ has external "C++" linkage unless otherwise specified.

Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

### 2.5.3 Constraints on programs

Restrictions on C++ programs that use the facilities of the Embedded Warrior Library for C++.

#### 2.5.3.1 Reserved Names
EWL reserves certain sets of names and function signatures for its implementation.

Names that contain a double underscore (_ _) or begins with an underscore followed by an upper-case letter is reserved to the EWL library for its use.

Names that begin with an underscore are reserved to the library for use as a name in the global namespace.

User code can safely use macros that are all uppercase characters and underscores, except for leading underscores. Library code will either be in namespace std or in namespace Metrowerks. Implementation details in namespace std will be prefixed by a double underscore or an underscore followed by an uppercase character. Implementation details in namespace Metrowerks are nested in a nested namespace, for example:

```
Metrowerks::details
```

### 2.5.3.2 External Linkage

Each name from the Embedded Warrior Library for C declared with external linkage is reserved to the implementation for use as a name with extern "C" linkage, both in namespace std and in the global namespace.

### 2.5.3.3 Headers

The behavior of any header file with the same name as a Embedded Warrior Library public or private header is undefined.

### 2.5.3.4 Derived classes

Virtual member function signatures defined for a base class in the C++ Standard Library may be overridden in a derived class defined in the program.

### 2.5.3.5 Replacement Functions
If replacement definition occurs prior to the program startup then replacement functions are allowed.

A C++ program may provide the definition for any of eight dynamic memory allocation function signatures declared in header `<new>`.

**Listing: Dynamic Memory Allocators**

```cpp
operator new(size_t)
operator new(size_t, const std::nothrow_t&)
operator new[](size_t)
operator new[](size_t, const std::nothrow_t&)
operator delete(void*)
operator delete(void*, const std::nothrow_t&)
operator delete[](void*)
operator delete[](void*, const std::nothrow_t&)
```

### 2.5.3.6 Handler functions

The EWL C++ library provides default versions of the following handler functions:

- `unexpected_handler`
- `terminate_handler`

A C++ program may install different handler functions during execution, by supplying a pointer to a function defined in the program or the library as an argument to:

- `set_new_handler`
- `set_unexpected`
- `set_terminate`

### 2.5.3.7 Other functions

In certain cases the EWL C++ depends on components supplied by a C++ program. If these components do not meet their requirements, the behavior is undefined.

### 2.5.3.8 Function arguments

If a C++ library function is passed incorrect but legal arguments the behavior is undefined.
2.5.4 Conforming Implementations

EWL is an ANSI/ISO Conforming implementation as described by the ANSI/ISO Standards in section 17.4.4
Chapter 3
Language Support Library

This chapter describes the implicit functions and temporary objects that may be generated during the execution of some C++ programs. It also contains information about the headers for those function, objects and defined types.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- **Types** covers predefined types
- **Implementation properties** covers implementation defined properties
- **Start and termination** covers functions used for starting and termination of a program
- **Dynamic Memory Management** covers operators used for dynamic allocation and release of memory.
- **Type identification** covers objects and functions used for runtime type identification.
- **Exception Handling** covers objects and functions used for exception handling and errors in exception handling.
- **Other Runtime Support** covers variations of the standard C library support functions.

### 3.1 Types

The header `<cstddef>` contains the same types and definitions as the standard C `stddef.h` with the changes shown in the following table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>The macro NULL is an implementation-defined C++ constant value. EWL defines this as 0L.</td>
</tr>
<tr>
<td>offsetof</td>
<td>This macro accepts a restricted set of type arguments that shall be a POD structure or a POD union. The result of applying the offsetof macro to a field that is a static data member or a function member is undefined.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 3-1. Header <cstddef> (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrdiff_t</td>
<td>No change from standard C. An signed integral type large enough to hold the difference between two pointers.</td>
</tr>
<tr>
<td>size_t</td>
<td>No change from standard C. An unsigned integral type large enough to hold the result of the sizeof operator.</td>
</tr>
</tbody>
</table>

3.2 Implementation properties

The headers <limits>, <climits>, and <cfloat> supply implementation dependent characteristics for fundamental types.

3.2.1 Numeric limits

The numeric_limits component provides a C++ program with information about various properties of the implementation's representation of the fundamental types.

Specializations including floating point and integer types are provided.

- The member is_specialized shall be true for specializations of numeric_limits.
- Members declared static const in the numeric_limits template specializations are usable as integral constant expressions.
- Non-fundamental standard types, do not have specializations.

All static members shall be provided but they do not need to be used.

3.2.2 is_specialized

The data member for distinguishing specializations. The default value is false.

```cpp
static const bool is_specialized = false;
```

3.2.3 min


The minimum positive normalized value is returned.

\[ \text{static T min() throw();} \]

### 3.2.4 max

The minimum finite value for floating point types with denormalization.

\[ \text{static T max() throw();} \]

**Remarks**

The maximum positive normalized value is returned.

### 3.2.5 digits

Designates the number of non-signed digits that can be represented for integral types. The number of radix digits in the mantissa for floating point types.

\[ \text{static const int digits = 0;} \]

### 3.2.6 is_signed

True if the number is signed.

\[ \text{static const bool is_signed = false;} \]

### 3.2.7 is_integer

True if the number is an integer.
static const bool is_integer = false;

3.2.8 is_exact

True if the number is exact.

static const bool is_exact = false;

Remarks
All integer types are exact, but not all floating point types are exact.

3.2.9 radix

Specifies the base or radix of the exponent of a floating point type or base of an integral type.

static const int radix = 0;

3.2.10 epsilon

The difference between 1 and the least value greater than 1.

static T epsilon() throw();

3.2.11 round_error

A function to measure the rounding error.

static T round_error() throw();
Remarks
Returns the maximum rounding error.

3.2.12 min_exponent

Holds the minimum exponent so that the radix raised to one less than this would be normalized.

    static const int min_exponent;

3.2.13 min_exponent10

Stores the minimum negative exponent that 10 raised to that power would be a normalized floating point type.

    static const int min_exponent10 = 0;

3.2.14 max_exponent

The maximum positive integer so that the radix raised to the power one less than this is representable.

    static const int max_exponent = 0;

3.2.15 max_exponent10

The maximum positive integer so that the 10 raised to this power is representable.

    static const int max_exponent10 = 0;
3.2.16 has_infinity

True if the type is positive for infinity.

    static const bool has_infinity = false;

3.2.17 has_quiet_NaN

True if the type has a quiet "Not a Number".

    static const bool has_quiet_NaN = false;

3.2.18 has_signaling_NaN

True if the type is a signaling "Not a Number".

    static const bool has_signaling_NaN = false;

3.2.19 has_denorm

Distinguishes if the floating point number has the ability to be denormalized.

    static const float_denorm_style has_denorm = denorm_absent;

Remarks

The static variable has_denorm equals denorm_present if the type allows denormalized values. The variable has_denorm equals denorm_absent if the type does not allow denormalized values. The variable has_denorm equals denorm_indeterminate if the type is indeterminate for denormalized values.
3.2.20 has_denorm_loss

Is true if there is a loss of accuracy because of a denormalization loss.

    static const bool has_denorm_loss = false;

3.2.21 infinity

Determines a positive infinity.

    static T infinity() throw();

Remarks
Returns a positive infinity if available.

3.2.22 quiet_NaN

Determines if there is a quiet "Not a Number".

    static T quiet_NaN() throw();

Remarks
Returns a quiet "Not a Number" if available.

3.2.23 signaling_NaN

Determines if there is a signaling "Not a Number".

    static T signaling_NaN() throw();
Remarks
Returns a signaling "Not a Number" if available.

3.2.24  denorm_min

Determines the minimum positive denormalized value.

    static T denorm_min() throw();

Remarks
Returns the minimum positive denormalized value.

3.2.25  is_iec559

The values is true if and only if the type adheres to IEC 59 standard

    static const bool is_iec559 = false;

3.2.26  is_bounded

The value is true if the set of values representable by the type is finite.

    static const bool is_bounded = false;

Remarks
All predefined data types are bounded.

3.2.27  is_modulo
This value is true if the type is modulo. A type is modulo if it is possible to add two positive numbers and have a result that wraps around to a third number that is less.

```cpp
static const bool is_modulo = false;
```

**Remarks**

This value is generally true for unsigned integral types and false for floating point types.

### 3.2.28 traps

The value is true if trapping is implemented for the type.

```cpp
static const bool traps = false;
```

### 3.2.29 tinyness_before

This value is true if tinyness is detected before rounding.

```cpp
static const bool tinyness_before = false;
```

### 3.2.30 round_style

This value is the rounding style as a type float_round_style.

```cpp
static const float_round_style round_style =
    round_toward_zero;
```

**Remarks**

See Also [Floating Point Rounding Styles](#)
3.2.31 Type float_round_style

An enumerated type in std namespace that is used to determine the characteristics for rounding floating point numbers.

<table>
<thead>
<tr>
<th>Enumerated Type</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>round_indeterminate</td>
<td>-1</td>
<td>The rounding is indeterminable</td>
</tr>
<tr>
<td>round_toward_zero</td>
<td>0</td>
<td>The rounding is toward zero</td>
</tr>
<tr>
<td>round_to_nearest</td>
<td>1</td>
<td>Round is to the nearest value</td>
</tr>
<tr>
<td>round_toward_infinity</td>
<td>2</td>
<td>The rounding is to infinity</td>
</tr>
<tr>
<td>round_toward_neg_infinity</td>
<td>3</td>
<td>The rounding is to negative infinity</td>
</tr>
</tbody>
</table>

3.2.32 Type float_denorm_style

The presence of denormalization is represented by the std namespace enumerated type float_denorm_style.

<table>
<thead>
<tr>
<th>Enumerated Type</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>denorm_indeterminate</td>
<td>-1</td>
<td>Denormalization is indeterminable</td>
</tr>
<tr>
<td>denorm_absent</td>
<td>0</td>
<td>Denormalization is absent</td>
</tr>
<tr>
<td>denorm_present</td>
<td>1</td>
<td>Denormalization is present</td>
</tr>
</tbody>
</table>

3.2.33 numeric_limits specializations

All members have specializations but these values are not required to be meaningful. Any value that is not meaningful is set to 0 or false.

C Library

The contents of <climits> are the same as standard C's limits.h and the contents of <cfloat> are the same as standard C's float.h.

<table>
<thead>
<tr>
<th>Enumerated Type</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_MAX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 3-4. Header <climits> (continued)

<table>
<thead>
<tr>
<th>INT_MIN</th>
<th>LONG_MAX</th>
<th>LONG_MIN</th>
<th>MB_LEN_MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHAR_MAX</td>
<td>SCHAR_MIN</td>
<td>SHRT_MAX</td>
<td>SHRT_MIN</td>
</tr>
<tr>
<td>UCHAR_MAX</td>
<td>UINT_MAX</td>
<td>ULONG_MAX</td>
<td>USHRT_MAX</td>
</tr>
</tbody>
</table>

The header <cfloat> is the same as standard C float.h

Table 3-5. Header <cfloat>

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBL_DIG</td>
<td>DBL_EPSILON</td>
<td>DBL_MANT_DIG</td>
</tr>
<tr>
<td>DBL_MAX</td>
<td>DBL_MAX_10_EXP</td>
<td>DBL_MAX_EXP</td>
</tr>
<tr>
<td>DBL_MIN</td>
<td>DBL_MIN_10_EXP</td>
<td>DBL_MIN_EXP</td>
</tr>
<tr>
<td>FLT_DIG</td>
<td>FLT_EPSILON</td>
<td>FLT_MANT_DIG</td>
</tr>
<tr>
<td>FLT_MAX</td>
<td>FLT_MAX_10_EXP</td>
<td>FLT_MAX_EXP</td>
</tr>
<tr>
<td>FLT_MIN</td>
<td>FLT_MIN_10_EXP</td>
<td>FLT_MIN_EXP</td>
</tr>
<tr>
<td>FLT_RADIX</td>
<td>FLT_ROUNDS</td>
<td>LDBL_DIG</td>
</tr>
<tr>
<td>LDBL_EPSILON</td>
<td>LDBL_MANT_DIG</td>
<td>LDBL_MAX</td>
</tr>
<tr>
<td>LDBL_MAX_10_EXP</td>
<td>LDBL_MAX_EXP</td>
<td>LDBL_MIN</td>
</tr>
<tr>
<td>LDBL_MIN_10_EXP</td>
<td>LDBL_MIN_EXP</td>
<td></td>
</tr>
</tbody>
</table>

3.3 Start and termination

The header <cstdlib> has the same functionality as the standard C header stdlib.h in regards to start and termination functions except for the functions and macros as described below.

Table 3-6. Start and Termination Differences

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXIT_FAILURE</td>
<td>1</td>
<td>This macro is used to signify a failed return</td>
</tr>
<tr>
<td>EXIT_SUCCESS</td>
<td>0</td>
<td>This macro is used to signify a successful return</td>
</tr>
</tbody>
</table>

The return from the `main` function is ignored on the Macintosh operating system and is returned using the native event processing method on other operating systems.
3.3.1 abort

Terminates the Program with abnormal termination.

abort(void)

Remarks
The program is terminated without executing destructors for objects of automatic or static storage duration and without calling the functions passed to atexit.

3.3.2 atexit

The atexit function registers functions to be called when exit is called in normal program termination.

extern "C" int atexit(void (* f)(void))
extern "C++" int atexit(void (* f)(void))

Remarks
If there is no handler for a thrown exception terminate is called. The registration of at least 32 functions is allowed.

- Functions registered with atexit are called in reverse order.
- A function registered with atexit before an object of static storage duration will not be called until the object's destruction.
- A function registered with atexit after an object of static storage duration is initialized will be called before the object's destruction.

The atexit() function returns zero if the registration succeeds, non zero if it fails.

3.3.3 exit

Terminates the program with normal cleanup actions.

exit(int status)

Remarks
The function `exit()` has additional behavior in the following order:

- Objects with static storage duration are destroyed and functions registered by calling `atexit` are called.
- Objects with static storage duration are destroyed in the reverse order of construction. If the `main()` function contains no automatic objects control can be transferred to `main()` if an exception thrown is caught in `main()`.
- Functions registered with `atexit` are called.
- All open C streams with unwritten buffered data are flushed, closed, including streams associated with `cin` and `cout`. All `tmpfile()` files are removed.
- Control is returned to the host environment.

If status is zero or `EXIT_SUCCESS`, a successful termination is returned to the host environment.

If status is `EXIT_FAILURE`, an unsuccessful termination is returned to the host environment.

Otherwise the status returned to the host environment is implementation-defined.

### 3.4 Dynamic Memory Management

The header `<new>` defines procedures for the management of dynamic allocation and error reporting of dynamic allocation errors.

#### 3.4.1 Storage Allocation and Deallocation

This clause covers storage allocation and deallocation functions and error management.

#### 3.4.2 Single Object Forms

Dynamic allocation and freeing of single object data types.
3.4.2.1 operator new

Dynamically allocates signable objects.

```cpp
void* operator new (std::size_t size) throw(std::bad_alloc);
void* operator new (std::size_t size, const std::nothrow_t&) throw();
```

Remarks

The nothrow version of `new` returns a null pointer on failure. The normal version throws a `bad_alloc` exception on error.

Returns a pointer to the allocated memory.

3.4.2.2 operator delete

Frees memory allocated with operator new.

```cpp
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

3.4.3 Array Forms

Dynamic allocation and freeing of array based data types.

3.4.3.1 operator new[]

Used for dynamic allocation or array based data types.

```cpp
void* operator new[]
(std::size_t size) throw(std::bad_alloc);
void* operator new[]
(std::size_t size, const std::nothrow_t&) throw();
```
Remarks
The default `operator new` will throw an exception upon failure. The nothrow version will return `NULL` upon failure.

3.4.3.2 `operator delete[]`

Operator `delete[]` is used in conjunction with `operator new[]` for array allocations.

```cpp
void operator delete[]
(void* ptr) throw();
void operator delete[]
(void* ptr, const std::nothrow_t&) throw();
```

3.4.4 Placement Forms

Placement operators are reserved and may not be overloaded by a C++ program.

3.4.4.1 Placement `operator new`

Allocates memory at a specific memory address.

```cpp
void* operator new (std::size_t size, void* ptr) throw();
void* operator new[](std::size_t size, void* ptr) throw();
```

3.4.4.2 Placement `operator delete`

The placement delete operators are used in conjunction with the corresponding placement new operators.

```cpp
void operator delete (void* ptr, void*) throw();
void operator delete[](void* ptr, void*) throw();
```
3.4.5 Storage Allocation Errors

C++ provides for various objects, functions and types for management of allocation errors.

3.4.5.1 Class Bad_alloc

A class used to report a failed memory allocation attempt.

3.4.5.2 Constructor

Constructs a bad_alloc object.

```cpp
bad_alloc() throw();
bad_alloc(const bad_alloc&) throw();
```

3.4.5.3 Assignment Operator

Assigns one bad_alloc object to another bad_alloc object.

```cpp
bad_alloc& operator=(const bad_alloc&) throw();
```

3.4.5.4 destructor

Destroys the bad_alloc object.

```cpp
virtual ~bad_alloc() throw();
```

3.4.5.5 what
An error message describing the allocation exception.

```cpp
virtual const char* what() const throw();
```

Returns a null terminated byte string "bad_alloc".

### 3.4.5.6 type new_handler

The type of a handler function that is called by operator new or operator new[].

```cpp
typedef void (*new_handler)();
```

If new requires more memory allocation, the new_handler will:

- Allocate more memory and return.
- Throw an exception of type bad_alloc or bad_alloc derived class.
- Either call abort or exit.

#### 3.4.5.7 set_new_handler

Sets the new handler function.

```cpp
new_handler set_new_handler
(new_handler new_p) throw();
```

Returns zero on the first call and the previous new_handler upon further calls.

### 3.5 Type identification

The header `<typeinfo>` defines three types for type identification and type identification errors.

The three classes are:

- Class type_info
- Class bad_cast
- Class bad_typeid
3.5.1 Class type_info

Class type_info contains functions and operations to obtain information about a type.

3.5.1.1 Constructors

A private constructor is included to prevent copying of this object.

    type_info(const type_info& rhs);

3.5.1.2 Assignment Operator

A private assignment is included to prevent copying of this object.

    type_info& operator=(const type_info& rhs);

3.5.1.3 operator==

Returns true if types are the same.

    bool operator==(const type_info& rhs) const;

Returns true if the objects are the same type.

3.5.1.4 operator!=

Compares for inequality.

    bool operator!=(const type_info& rhs) const;

Returns true if the objects are not the same type.
3.5.1.5 before

Is true if this object precedes the argument in collation order.

```cpp
bool before(const type_info& rhs) const;
```

Returns true if the this pointer precedes the argument the collation order.

3.5.1.6 name

Returns the name of the class.

```cpp
const char* name() const;
```

3.5.2 Class bad_cast

A class for exceptions thrown in runtime casting.

3.5.2.1 Constructors

Constructs an object of class bad_cast.

```cpp
bad_cast() throw();
bad_cast(const bad_cast&) throw();
```

3.5.2.2 Assignment Operator

Copies an object of class bad_cast.

```cpp
bad_cast& operator=(const bad_cast&) throw();
```
3.5.2.3 what

An error message describing the casting exception.

```
virtual const char* what() const throw();
```

Returns the null terminated byte string "bad_cast".

3.5.3 Class bad_typeid

Defines a type used for handling typeid exceptions.

3.5.3.1 Constructors

Constructs an object of class bad_typeid.

```
bad_typeid() throw();
bad_typeid(const bad_typeid&) throw();
```

3.5.3.2 Assignment Operator

Copies a class bad_typeid object.

```
bad_typeid& operator=(const bad_typeid&) throw();
```

3.5.3.3 what

An error message describing the typeid exception.

```
virtual const char* what() const throw();
```
Returns the null terminated byte string "bad_typeid".

3.6 Exception Handling

The header <exception> defines types and procedures necessary for the handling of exceptions.

3.6.1 Class exception

A base class for objects thrown as exceptions.

3.6.1.1 Constructors

Constructs an object of the exception class.

```cpp
exception() throw();
exception(const exception&) throw();
```

3.6.1.2 Assignment Operator

Copies an object of exception class.

```cpp
exception& operator=(const exception&) throw();
```

3.6.1.3 Destructor

Destroys an exception object.

```cpp
virtual ~exception() throw();
```
3.6.1.4 what

An error message describing the exception.

    virtual const char* what() const throw();

Returns the null terminated byte string "exception".

3.6.2 Violating Exception Specifications

Defines objects used for exception violations.

3.6.2.1 Class bad_exception

A type used for information and reporting of a bad exceptions.

3.6.2.1.1 Constructors

Constructs an object of class bad_exception.

    bad_exception() throw();
    bad_exception(const bad_exception&) throw();

3.6.2.1.2 Assignment Operator

Copies an object of class bad_exception

    bad_exception& operator=
    (const bad_exception&) throw();

3.6.2.1.3 what

An error message describing the bad exception.
virtual const char* what() const throw();

Returns the null terminated byte string "bad_exception".

3.6.2.1.4 type unexpected_handler

A type of handler called by the unexpected function.

typedef void (*unexpected_handler)();

The unexpected_handler calls terminate().

3.6.2.1.5 set_unexpected

Sets the unexpected handler function.

unexpected_handler set_unexpected
(unexpected_handler f) throw();

Returns the previous unexpected_handler.

3.6.2.1.6 unexpected

Called when a function ends by an exception not allowed in the specifications.

void unexpected();

May be called directly by the program.

3.7 Abnormal Termination

Types and functions used for abnormal program termination.

3.7.1 type terminate_handler
A type of handler called by the function `terminate` when terminating an exception.

```c
typedef void (*terminate_handler)();
```

The `terminate_handler` calls `abort()`.

### 3.7.2 set_terminate

Sets the function for terminating an exception.

```c
terminate_handler set_terminate
(terminate_handler f) throw();
```

The `terminate_handler` shall not be a null pointer.

The previous `terminate_handler` is returned.

### 3.7.3 terminate

A function called when exception handling is abandoned.

```c
void terminate();
```

Exception handling may be abandoned by the implementation (for example the default handler) or may be called directly by the program (uncaught exception) among other reasons. These errors in the exception handling mechanism are handled using `terminate`.

### 3.7.4 uncaught_exception

Determines an uncaught exception.

```c
bool uncaught_exception();
```

Throwing an exception while `uncaught_exception` is true can result in a call of `terminate`.

Returns true if an exception is uncaught.
3.8 Other Runtime Support

The C++ headers `<cstdarg>`, `<csetjmp>`, `<ctime>`, `<csignal>` and `<cstdlib>` contain macros, types and functions that vary from the corresponding standard C headers.

### Table 3-7. Header `<cstdarg>`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>va_arg</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>va_end</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>va_start</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>va_list</td>
<td>A type used in C++ Runtime support</td>
</tr>
</tbody>
</table>

If the second parameter of `va_start` is declared with a function, array, reference type or with a type for which there is no parameter, the behavior is undefined.

### Table 3-8. Header `<csetjmp>`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setjmp</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>jmp_buf</td>
<td>A type used in C++ Runtime support</td>
</tr>
<tr>
<td>longjmp</td>
<td>A function used in C++ Runtime support</td>
</tr>
</tbody>
</table>

The function `longjmp` is more restricted than in the standard C implementation.

### Table 3-9. Header `<ctime>`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK_S_PER_SEC</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>clock_t</td>
<td>A type used in C++ Runtime support</td>
</tr>
<tr>
<td>clock</td>
<td>A function used in C++ Runtime support</td>
</tr>
</tbody>
</table>

If a signal handler attempts to use exception handling the result is undefined.

### Table 3-10. Header `<csignal>`

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>SIGILL</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>SIG_DFL</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>SIG_IGN</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>SIGINT</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>SIG_ERR</td>
<td>A macro used in C++ Runtime support</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 3-10. Header `<csignal>` (continued)

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sig_atomic_t</td>
<td>A macro used in C++ Runtime support</td>
</tr>
<tr>
<td>raise</td>
<td>A type used in C++ Runtime support</td>
</tr>
<tr>
<td>signal</td>
<td>A function used in C++ Runtime support</td>
</tr>
</tbody>
</table>

**NOTE**
All signal handlers should have C linkage.

### Table 3-11. Header `<cstdlib>`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getenv</td>
<td>A function used in C++ Runtime support</td>
</tr>
<tr>
<td>system</td>
<td>A function used in C++ Runtime support</td>
</tr>
</tbody>
</table>
Chapter 4
Diagnostics Library

This chapter describes objects and facilities used to report error conditions.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Exception Classes
- Assertions
- Error Numbers

4.1 Exception Classes

The library provides for exception classes for use with logic errors and runtime errors. Logic errors in theory can be predicted in advance while runtime errors can not. The header <stdexcept> predefines several types of exceptions for C++ error reporting.

There are nine exception classes.

- Class Logic_error
- Class domain_error
- Class Invalid_argument
- Class Length_error
- Class Out_of_range
- Class Runtime_error
- Class Range_error
- Class Overflow_error
- Class Underflow_error
4.1.1 Class logic_error

The `logic_error` class is derived from the `Class exception` and is used for exceptions that are detectable before program execution.

**Constructors**

```cpp
logic_error(const string& what_arg);
```

Constructs an object of class `logic_error`. Initializes `exception::what` to the `what_arg` argument.

4.1.2 Class domain_error

A derived class of logic error the `domain_error` object is used for exceptions of domain errors.

**Constructors**

```cpp
domain_error(const string& what_arg);
```

Constructs an object of `domain_error`. Initializes `exception::what` to the `what_arg` argument.

4.1.3 Class invalid_argument

A derived class of `logic_error` the `invalid_argument` is used for exceptions of invalid arguments.

**Constructors**

```cpp
invalid_argument(const string& what_arg);
```

Constructs an object of class `invalid_argument`. Initializes `exception::what` to the `what_arg` argument.
4.1.4 Class length_error

A derived class of logic_error the length_error is used to report exceptions when an object exceeds allowed sizes.

Constructors

\[
\text{length_error} \left( \text{const string& what_arg} \right);
\]

Constructs an object of class length_error. Initializes exception::what to the what_arg argument

4.1.5 Class out_of_range

A derived class of logic_error an object of out_of_range is used for exceptions for out of range errors.

Constructors

\[
\text{out_of_range} \left( \text{const string& what_arg} \right);
\]

Constructs an object of the class out_of_range. Initializes exception::what to the what_arg argument

4.1.6 Class runtime_error

Derived from the Class exception the runtime_error object is used to report errors detectable only during runtime.

Constructors

Constructs an object of the class runtime_error. Initializes exception::what to the what_arg argument

4.1.7 Class range_error
Derived from the `runtime_error` class, an object of `range_error` is used for exceptions due to runtime out of range errors.

**Constructors**

```cpp
runtime_error(const string& what_arg);
range_error(const string& what_arg);
```

Constructs an object of the class `range_error`. Initializes `exception::what` to the `what_arg` argument.

### 4.1.8 Class `overflow_error`

The `overflow_error` object is derived from the class `runtime_error` and is used to report arithmetical overflow errors.

**Constructors**

```cpp
overflow_error(const string& what_arg);
```

Constructs an object of the class `overflow_error`. Initializes `exception::what` to the `what_arg` argument.

### 4.1.9 Class `underflow_error`

The class `underflow_error` is derived from the class `runtime_error` and is used to report the arithmetical underflow error.

**Constructors**

```cpp
underflow_error(const string& what_arg);
```

Constructs an object of the class `underflow_error`. Initializes `exception::what` to the `what_arg` argument.

### 4.2 Assertions
The header `<cassert>` provides for the assert macro and is used the same as the standard C header `assert.h`

### 4.3 Error Numbers

The header `<cerrno>` provides macros: EDOM ERANGE and errno to be used for domain and range errors reported by using the errno facility. The `<cerrno>` header is used the same as standard C header `errno.h`
Chapter 5
General Utilities Libraries

This clause describes components used by other elements of the Standard C++ library. These components may also be used by C++ programs.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Requirements
- Utility Components
- Pairs
- Function objects
- Memory
- Template Class Auto_ptr
- C Library
- Date and Time

5.1 Requirements

This section describes the requirements for template arguments, types used to instantiate templates and storage allocators used as general utilities.

5.1.1 Equality Comparisons

The equality comparison operator is required. The (==) expression has a bool return type and specifies that for \( x = y \) and \( y = z \) that \( x \) will equal \( z \). In addition the reciprocal is also true. That is, if \( x = y \) then \( y \) equals \( x \). Also if \( x = y \) and \( y = z \) then \( z \) will be equal to \( x \).
5.1.2 Less Than Comparison

A less than operator is required. The \(<\) expression has a \texttt{bool} return type and states that if \(x < y\) that \(x\) is less than \(y\) and that \(y\) is not less than \(x\).

5.1.3 Copy Construction

A copy constructor for the general utilities library has the following requirements:

- If the copy constructor is \texttt{TYPE(t)} then the argument must be an equivalent of \texttt{TYPE}.
- If the copy constructor is \texttt{TYPE(const t)} then the argument must be the equivalent of \texttt{const TYPE}.
- \texttt{&T}, denotes the address of \texttt{T}.
- \texttt{&const T}, denotes the address of \texttt{const T}.

5.1.4 Default Construction

A default constructor is not necessary. However, some container class members may specify a default constructor as a default argument. In that case when a default constructor is used as a default argument there must be a default constructor defined.

5.1.5 Allocator Requirements

The general utilities library requirements include requirements for allocators. Allocators are objects that contain information about the container. This includes information concerning pointer types, the type of their difference, the size of objects in this allocation, also the memory allocation and deallocation information. All of the standard containers are parameterized in terms of allocators.
The allocator class includes the following members

Table 5-1. Allocator Members

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>A pointer to a type</td>
</tr>
<tr>
<td>const_pointer</td>
<td>A pointer to a const type</td>
</tr>
<tr>
<td>reference</td>
<td>A reference of a type</td>
</tr>
<tr>
<td>const_reference</td>
<td>A reference to a const type</td>
</tr>
<tr>
<td>value_type</td>
<td>A type identical to the type</td>
</tr>
<tr>
<td>size_type</td>
<td>An unsigned integer that can represent the largest object in the allocator</td>
</tr>
<tr>
<td>difference_type</td>
<td>A signed integer that can represent the difference between any two pointers in the allocator</td>
</tr>
<tr>
<td>rebind</td>
<td>The template member is effectively a typedef of the type to which the allocator is bound</td>
</tr>
<tr>
<td>address(type)</td>
<td>Returns the address of type</td>
</tr>
<tr>
<td>address(const type)</td>
<td>Returns the address of the const type</td>
</tr>
<tr>
<td>allocate(size)</td>
<td>Returns the allocation of size</td>
</tr>
<tr>
<td>allocate(size, address)</td>
<td>Returns the allocation of size at the address</td>
</tr>
<tr>
<td>max_size</td>
<td>The largest value that can be passed to allocate</td>
</tr>
<tr>
<td>Ax == Ay</td>
<td>Returns a bool true if the storage of each allocator can be deallocated by the other</td>
</tr>
<tr>
<td>Ax != Ay</td>
<td>Returns a bool true if the storage of each allocator cannot be deallocated by the other</td>
</tr>
<tr>
<td>T()</td>
<td>Constructs an instance of type</td>
</tr>
<tr>
<td>T x(y)</td>
<td>x is constructed with the values of y</td>
</tr>
</tbody>
</table>

Allocator template parameters must meet additional requirements

- All instances of an allocator are interchangeable and compare equal to each other
- Members must meet the requirements in Table 5-2

Implementation-defined allocators are allowed.

Table 5-2. The Typedef Members Requirements

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>T*</td>
</tr>
<tr>
<td>const_pointer</td>
<td>T const*</td>
</tr>
<tr>
<td>size_type</td>
<td>size_t</td>
</tr>
<tr>
<td>difference_type</td>
<td>ptrdiff_t</td>
</tr>
</tbody>
</table>
5.2 Utility Components

This sub-clause contains some basic template functions and classes that are used throughout the rest of the library.

5.2.1 Operators

The Standard C++ library provides general templatized comparison operators that are based on operator== and operator<.

5.2.1.1 operator!=

This operator determines if the first argument is not equal to the second argument.

\[
\text{template <class T> bool operator!=(const T& x, const T& y);}\]

5.2.1.2 operator>

This operator determines if the first argument is less than the second argument.

\[
\text{template <class T> bool operator>(const T& x, const T& y);}\]

5.2.1.3 operator<=

This operator determines if the first argument is less than or equal to the second argument.

\[
\text{template <class T> bool operator<=(const T& x, const T& y);}\]
5.2.1.4 operator>=

This operator determines if the first argument is greater than or equal to the second argument.

```
template <class T> bool operator>=(const T& x, const T& y);
```

5.3 Pairs

The utility library includes support for paired values.

5.3.1 Constructors

The pair class contains various constructors to fit each pairs needs.

```
pair();
```

Initializes its members as with default type constructors.

```
template<class U, class V> pair(const pair<U, V>& p);
```

Initializes and does any implicit conversions if necessary.

5.3.2 operator ==

The pair equality operator returns true if each pair argument is equal to the other.

```
template <class T1, class T2>
bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

5.3.3 operator <
Function objects

The pair less than operator returns true if the second pair argument is less that the first pair argument.

```cpp
template <class T1, class T2> bool operator <
    const pair<T1, T2>& x, const pair<T1, T2>& y);
```

5.3.4 make_pair

Makes a pair of the two arguments.

```cpp
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

Remarks

Returns a pair of the two arguments.

5.4 Function objects

Function objects have the operator() defined and used for more effective use of the library.

When a pointer to a function would normally be passed to an algorithm function the library is specified to accept an object with operator() defined. The use of function objects with function templates increases the power and efficiency of the library.

Struct Unary_function and Struct Binary_function classes are provided to simplify the typedef of the argument and result types.

NOTE

In order to manipulate function objects that take one or two arguments it is required that their function objects provide the defined types. If the function object takes one argument then argument_type and result_type are defined. If the function object takes two arguments then the first_argument_type, second_argument_type, and result_type must be defined.
5.4.1 Arithmetic operations

The utility library provides function object classes with operator() defined for the arithmetic operations.

5.4.1.1 plus

 Adds the first and the second and returns that sum.

    template <class T> struct plus : binary_function<T,T,T> {
       T operator()(const T& x, const T& y) const;
    };

Remarks

Returns x plus y.

5.4.1.2 minus

Subtracts the second from the first and returns the difference.

    template <class T> struct minus : binary_function<T,T,T> {
       T operator()(const T& x, const T& y) const;
    };

Remarks

Returns x minus y.

5.4.1.3 multiplies

Multiplies the first times the second and returns the resulting value.

    template <class T> struct multiplies : binary_function<T,T,T> {
       T operator()(const T& x, const T& y) const;
    };

Remarks
Returns $x$ multiplied by $y$.

5.4.1.4 divides

Divides the first by the second and returns the resulting value.

```cpp
template <class T> struct divides : binary_function<T,T,T>
{
    T operator()(const T& x, const T& y) const;
};
```

Remarks

Returns $x$ divided by $y$.

5.4.1.5 modulus

Determines the modulus of the first by the second argument and returns the result.

```cpp
template <class T> struct modulus : binary_function<T,T,T>
{
    T operator()(const T& x, const T& y) const;
};
```

Remarks

Returns $x$ modulus $y$.

5.4.1.6 negate

This function returns the negative value of the argument.

```cpp
template <class T> struct negate : unary_function<T,T>
{
    T operator()(const T& x) const;
};
```

Remarks

Returns the negative of $x$.

5.4.2 Comparisons
The utility library provides function object classes with operator() defined for the comparison operations.

NOTE
For the greater, less, greater_equal and less_equal template classes specializations for pointers yield a total order.

5.4.2.1 equal_to

Returns true if the first argument is equal to the second argument.

```cpp
template <class T> struct equal_to :
  binary_function<T,T,bool>
{
  bool operator()(const T& x, const T& y) const;
};
```

Remarks
Returns true if x is equal to y.

5.4.2.2 not_equal_to

Returns true if the first argument is not equal to the second argument.

```cpp
template <class T> struct not_equal_to :
  binary_function<T,T,bool>
{
  bool operator()(const T& x, const T& y) const;
};
```

Remarks
Returns true if x is not equal to y.

5.4.2.3 greater

Returns true if the first argument is greater than the second argument.

```cpp
template <class T> struct greater : binary_function<T,T,bool>
{
  bool operator()(const T& x, const T& y) const;
};
```

Remarks
Returns true if $x$ is greater than $y$.

### 5.4.2.4 less

Returns true if the first argument is less than the second argument.

```cpp
template <class T> struct less : binary_function<T,T,bool>
{
    bool operator()(const T& x, const T& y) const;
};
```

**Remarks**

Returns true if $x$ is less than $y$.

### 5.4.2.5 greater_equal

Returns true if the first argument is greater than or equal to the second argument.

```cpp
template <class T> struct greater_equal :
    binary_function<T,T,bool>
{
    bool operator()(const T& x, const T& y) const;
};
```

**Remarks**

Returns true if $x$ is greater than or equal to $y$.

### 5.4.2.6 less_equal

Returns true if the first argument is less than or equal to the second argument.

```cpp
template <class T> struct less_equal :
    binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

**Remarks**

Returns true if $x$ is less than or equal to $y$. 
5.4.3 Logical operations

The utility library provides function object classes with operator() defined for the logical operations.

5.4.3.1 logical_and

Returns true if the first and the second argument are true.

```cpp
template <class T> struct logical_and :
    binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

Remarks

Returns true if x and y are true.

5.4.3.2 logical_or

Returns true if the first or the second argument are true.

```cpp
template <class T> struct logical_or :
    binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

Remarks

Returns true if the x or y are true.

5.4.3.3 logical_not

Returns true if the argument is zero

```cpp
template <class T> struct logical_not :
    unary_function<T,bool> {
    bool operator()(const T& x) const;
};
```

Remarks

Returns true if x is equal to zero.
5.4.4 Negators

The utility library provides negators `not1` and `not2` that returns the complement of the unary or binary predicate.

A predicate is an object that takes one or two arguments and returns something convertible to bool.

5.4.4.1 Unary_negate

In the template class `unary_negate` the operator() returns the compliment of the predicate argument.

`not1`

The template function `not1` returns the `unary_predicate` of the predicate argument.

```cpp
template <class Predicate>
unary_negate<Predicate>
not1(const Predicate& pred);
```

Remarks

Returns true if pred is not true.

5.4.4.2 binary_negate

In the template class `binary_negate` the operator() returns the compliment of the predicate arguments.

`not2`

The template function `not2` returns the `binary_predicate` of the predicate arguments.

```cpp
template <class Predicate>
binary_negate<Predicate>
not2(const Predicate& pred);
```

Remarks
Returns the compliment of the argument.

### 5.4.5 Binders

The binders classes, bind1st and bind2nd take a function object and a value and return a function object constructed out of the function bound to the value.

#### 5.4.5.1 Template class binder1st

The binders class bind1st takes a function object and a value and return a function object constructed out of the function bound to the value.

**Remarks**

The constructor initializes the operation.

#### 5.4.5.2 bind1st

Binds the first.

```cpp
  template <class Operation, class T>
  binder1st<Operation> bind1st(const Operation& op, const T& x);
```

**Remarks**

Binds the operation to the first argument type.

#### 5.4.5.3 Template class binder2nd

The binders class bind1st takes a function object and a value and return a function object constructed out of the function bound to the value.

**Remarks**

The constructor initializes the operation.
5.4.5.4  bind2nd

```cpp
template <class Operation, class T>
binder2nd<Operation> bind2nd
   (const Operation& op, const T& x);
```

**Remarks**
Binds the operation to the second argument type.

5.4.6  Adaptors for Pointers to Functions

Special adaptors for pointers to both unary and binary functions call them to work with function adaptors.

5.4.6.1  pointer_to_unary_function

```cpp
template <class Arg, class Result>
pointer_to_unary_function<Arg, Result>
   ptr_fun(Result (* f)(Arg));
```

**Remarks**
Returns a pointer for a unary function.

5.4.6.2  class pointer_to_binary_function

A class for a pointer used for binary binding.

5.4.6.3  pointer_to_binary_function
template <class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result
ptr_fun(Result (* f)(Arg1, Arg2));

Remarks
Returns a pointer for a binary function.

5.4.7 Adaptors for Pointers to Members

Adaptors for pointers to members are adaptors that allow you to call member functions for elements within a collection.

5.4.7.1 mem_fun_t

An adaptor for pointers to member functions.

    template<class S, class T>
    mem_fun_t<S,T,A> : public unary_function<T*, S>
    explicit mem_fun(S (T::*p)());

Remarks
The constructor for mem_fun_t calls the member function that is initialized with using a given pointer argument and an appropriate additional argument.

5.4.7.2 mem_fun1_t

A class for binding a member function.

    template<class S, class T, class A>
    class mem_fun1_t : public binary_function<T*, A, S>
    explicit mem_fun1_t(S (T::*p)(A));

Remarks
The constructor for mem_fun1_t calls the member function that it is initialized with using a given a pointer argument and an appropriate additional argument.
5.4.7.3  mem_fun

A function adapter for member functions

\begin{verbatim}
template<class S, class T> mem_fun_t<S,T>
    mem_fun(S (T::*f)());

template<class S, class T, class A>
    mem_fun(S (T::*f)(A));
\end{verbatim}

**Remarks**

The function returns an object through which a function can be called.

5.4.7.4  mem_fun_ref_t

A function adaptor for function reference objects.

\begin{verbatim}
template<class S, class T>
    class mem_fun_ref_t : public unary_function<T, S>
        explicit mem_fun_ref_t(S (T::*p)());
\end{verbatim}

**Remarks**

The function `mem_fun_ref_t` calls the member function reference it is initialized with using a given a reference argument.

5.4.7.5  mem_fun1_ref_t

A function adaptor for a member to function reference object.

\begin{verbatim}
template<class S, class T, class A>
    class mem_fun1_ref_t : public binary_function<T,A, S>
        explicit mem_fun1_ref_t(S (T::*p)(A));
\end{verbatim}

**Remarks**

The constructor for `mem_fun1_ref_t` calls the member function that it is initialized with a given a reference argument and an additional argument of the appropriate type.
5.4.7.6  mem_fun_ref

A function adaptor for a template member references.

```cpp
template<class S, class T> mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)(A));
template<class S, class T, class A> mem_fun1_ref_t<S, T, A> mem_fun_ref(S (T::*f)(A));
```

**Remarks**

The template function `mem_fun_ref` returns an object through which `X::*f` can be called given a reference to an `X` followed by the argument required for `f`.

5.4.7.7  const_mem_fun_t

A function adaptor for a constant member.

```cpp
template<class S, class T> class const_mem_fun_t : public unary_function<T*, S>
    explicit const_mem_fun(S (T::*p)() const);
```

**Remarks**

Provides a constant member to function object.

The constructor for `const_mem_fun_t` calls the member function that it is initialized with using a given pointer argument.

5.4.7.8  const_mem_fun1_t

A const to member function object type.

```cpp
template<class S, class T, class A> const_mem_fun1_t : public binary_function<T,A,S>
    explicit mem_fun_fun1_t(S (T::*p)(A) const);
```

**Remarks**

The constructor for `const_mem_fun1_t` calls the member function that it is initialized with using a given a pointer argument and an additional argument of the appropriate type.
5.4.7.9 const_mem_fun_ref_t

A function adaptor for a constant member reference.

```
template<class S, class T>
class const_mem_fun_ref_t<S,T> : public unary_function<T,S>

explicit const_mem_fun_ref_t( S (T::*p) () const);
```

Remarks

The template functions `mem_fun_ref` returns an object through which `X::f` can be called. The constructor for `const_mem_fun_ref_t` calls the member function that it is initialized with using a given a reference argument.

5.4.7.10 const_mem_fun1_ref_t

A constant member to function reference adaptor object.

```
template<class S, class T, class A>
class const_mem_fun1_ref_t<S,T>: public
binary_function<T,A,S>

explicit const_mem_fun1_ref_t( S (T::*p) (A) const);
```

Remarks

The constructor for `const_mem_fun1_ref_t` calls the member function it is initialized with using a given a reference argument and an additional argument of the appropriate type. The template functions `mem_fun_ref` returns an object through which `X::f` can be called.

5.5 Memory

The header `<memory>` includes functions and classes for the allocation and deallocation of memory.
5.5.1 allocator members

Members of the allocator class.

5.5.1.1 address

Determine the address of the allocation.

    pointer address(reference x) const;
    const_pointer address(const_reference x) const;

Remarks

Returns the address of the allocation.

5.5.1.2 allocate

Create an allocation and return a pointer to it.

    pointer allocate(size_type n, allocator<void>::const_pointer
                   hint=0);

Remarks

A pointer to the initial element of an array of storage.

Allocate throw a bad_alloc exception if the storage cannot be obtained.

5.5.1.3 deallocate

Remove an allocation from memory.

    void deallocate(pointer p, size_type n);

Deallocates the storage referenced by p.
5.5.1.4  max_size

Determines the Maximum size for an allocation.

    size_type max_size() const throw();

Remarks
Returns the largest size of memory that may be.

5.5.1.5  construct

Allocates an object and initializes it with a value.

    void construct(pointer p, const_reference val);

Remarks
A pointer to the allocated memory is returned.

5.5.1.6  destroy

Destroys the memory allocated

    void destroy(pointer p);

5.5.2  allocator globals

Provides globals operators in memory allocation.

5.5.2.1  operator==

Equality operator.
template <class T1, class T2> bool operator==(const allocator<T1>&, const allocator<T2>&) throw();

Remarks
Returns true if the arguments are equal.

5.5.2.2 operator!=

Inequality operator.

template <class T1, class T2> bool operator!=(const allocator<T1>&, const allocator<T2>&) throw();

Remarks
Returns true if the arguments are not equal.

5.5.3 Raw storage iterator

A means of storing the results of un-initialized memory.

NOTE
The formal template parameter OutputIterator is required to have its operator* return an object for which operator& is defined and returns a pointer to T, and is also required to satisfy the requirements of an output iterator.

5.5.3.1 Constructors

A constructor for the raw_storage_iterator class.

raw_storage_iterator(OutputIterator x);

Remarks
Initializes the iterator.

### 5.5.3.2 operator *

A dereference operator.

```cpp
raw_storage_iterator<OutputIterator,T>&
operator*();
```

**Remarks**

The dereference operator return `*this`.

### 5.5.3.3 operator=

The `raw_storage_iterator` assignment operator.

```cpp
raw_storage_iterator<OutputIterator,T>&
operator=(const T& element);
```

**Remarks**

Constructs a value from `element` at the location to which the iterator points.

A reference to the iterator.

### 5.5.3.4 operator++

Post and Pre-increment operators for `raw_storage_iterator`.

```cpp
raw_storage_iterator<OutputIterator,T>&
operator++();   // Pre-increment
raw_storage_iterator<OutputIterator,T>
operator++(int);   // Post-increment
```

**Remarks**

Increments the iterator. The post-increment operator returns the old value of the iterator.

The pre-increment operator returns the updated value.
5.5.4  Temporary buffers

Methods for storing and retrieving temporary allocations.

5.5.4.1  get_temporary_buffer

Retrieves a pointer to store temporary objects.

```cpp
template <class T> pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
```

Remarks

Returns an address for the buffer and its size or zero if unsuccessful.

5.5.4.2  return_temporary_buffer

Deallocation for the get_temporary_buffer procedure.

```cpp
template <class T>
void return_temporary_buffer(T* p);
```

Remarks

The buffer must have been previously allocated by get_temporary_buffer.

5.5.5  Specialized Algorithms

Algorithm necessary to fulfill iterator requirements.

5.5.5.1  uninitialized_copy

An uninitialized copy.
template <class InputIterator, 
        class ForwardIterator>
ForwardIterator uninitialized_copy
        (InputIterator first, InputIterator last, ForwardIterator
         result);

Remarks

Returns a ForwardIterator to the result argument.

5.5.5.2 uninitialized_fill

An uninitialized fill.

template <class ForwardIterator, class T>
void uninitialized_fill
        (ForwardIterator first, 
         ForwardIterator last, const T& x);

5.5.5.3 uninitialized_fill_n

An uninitialized fill with a size limit.

template <class ForwardIterator, 
        class Size, class T>
void uninitialized_fill_n
        (ForwardIterator first, Size n, const T& x);

5.6 Template Class auto_ptr

The auto_ptr class stores a pointer to an object obtained using new and deletes that object when it is destroyed. For example when a local allocation goes out of scope.

The template auto_ptr_ref holds a reference to an auto_ptr, and is used by the auto_ptr conversions. This allows auto_ptr objects to be passed to and returned from functions.
NOTE

An auto_ptr owns the object it holds a pointer to. When copying an auto_ptr the pointer transfers ownership to the destination.

If more than one auto_ptr owns the same object at the same time the behavior of the program is undefined.

See the example of using std::auto_ptr and extension version for arrays in Using Auto_ptr

This extension can be turned on by uncommenting the statement,

```
#define _EWL_ARRAY_AUTO_PTR in <ewlconfig>. No recompile of the C++ lib is necessary, but do rebuild any precompiled headers when making this change.
```

The functionality provided by the extended std::auto_ptr is very similar to that provided by the newer Metrowerks::alloc_ptr found in <ewl_utility>.

Listing: Using Auto_ptr

```cpp
#include <iostream>
#include <memory>
using std::auto_ptr;
using std::_Array;
struct A
{
    A() {std::cout << "construct A\n";}
    virtual ~A() {std::cout << "destruct A\n";}
};
struct B
    : A
{
    B() {std::cout << "construct B\n";}
    virtual ~B() {std::cout << "destruct B\n";}
};
auto_ptr<B> source();
void sink_b(auto_ptr<B>);
void sink_a(auto_ptr<A>);
auto_ptr<B, _Array<B> > array_source();
void array_sink(auto_ptr<B, _Array<B> >);
auto_ptr<B>
```
template Class auto_ptr

source()
{
    return auto_ptr<B>(new B);
}

void
sink_b(auto_ptr<B>)
{
}

void
sink_a(auto_ptr<A>)
{
}

auto_ptr<B, _Array<B> >
array_source()
{
    return auto_ptr<B, _Array<B> >(new B [2]);
}

void
array_sink(auto_ptr<B, _Array<B> >)
{
}

int main()
{
    
    auto_ptr<B> b(new B);
    auto_ptr<B> b2(b);
    b = b2;
    auto_ptr<B> b3(source());
    auto_ptr<A> a(b);
    a = b3;
    b3 = source();
    sink_b(source());
    auto_ptr<A> a2(source());
    a2 = source();
}
sink_a(source());

auto_ptr<B, _Array<B> > b(new B[2]);
auto_ptr<B, _Array<B> > b2(b);
b = b2;
auto_ptr<B, _Array<B> > b3(array_source());
b3 = array_source();
array_sink(array_source());

// auto_ptr<A, _Array<A> > a(b3);  // Should not compile

5.6.1 auto_ptr constructors

Constructs an auto_ptr object.

explicit auto_ptr(X* p =0) throw();

auto_ptr(auto_ptr& a) throw();

template<class Y> auto_ptr(auto_ptr<Y>& a) throw();

5.6.2 operator =

An auto_ptr assignment operator.

template<class Y> auto_ptr& operator=(
auto_ptr<Y>& a) throw();

auto_ptr& operator=
(auto_ptr& a) throw();

Remarks

Returns the this pointer.
5.6.3 destructor

Destroys the auto_ptr object.

/auto_ptr() throw();

5.6.4 auto_ptr Members

Member of the auto_ptr class.

5.6.5 operator*

The de-reference operator.

X& operator*() const throw();

Remarks
Returns what the dereferenced pointer *this holds.

5.6.6 operator->(

The pointer dereference operator.

X* operator->() const throw();

Remarks
Returns what the pointer *this holds.
5.6.7  get

Gets the value that the pointer points to.

\[
X^* \text{ get()} \text{ const throw();}
\]

Remarks
Returns what the pointer \*this holds.

5.6.8  release

Releases the auto_ptr object.

\[
X^* \text{ release()} \text{ throw();}
\]

Remarks
Returns what the pointer \*this holds.

5.6.9  reset

Resets the auto_ptr to zero or another pointer.

\[
\text{void reset}(X^* \ p=0) \text{ throw();}
\]

5.6.10  auto_ptr conversions

Conversion functionality for the auto_ptr class for copying and converting.

5.6.10.1  Conversion Constructor
A conversion constructor.

    auto_ptr(auto_ptr_ref<X> r) throw();

5.6.10.2  operator auto_ptr_ref

Provides a convert to lvalue process.

    template<class Y> operator auto_ptr_ref<Y>() throw();

Remarks
Returns a reference that holds the this pointer.

5.6.11  operator auto_ptr

Releases the auto_ptr and returns the pointer held.

    template<class Y> operator auto_ptr<Y>() throw();

Remarks
Returns the pointer held.

5.7  C Library

The EWL C++ memory libraries use the C library memory functions. See the EWL C Reference for <stdlib.h> functions calloc, malloc, free, realloc for more information.

5.8  Date and Time

The header <ctime> has the same contents as the Standard C library header <time.h> but within namespace std.
Chapter 6
Strings Library

This chapter is a reference guide to the ANSI/ISO String class that describes components for manipulating sequences of characters, where characters may be of type `char`, `wchar_t`, or of a type defined in a C++ program.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- **Character traits** defines types and facilities for character manipulations
- **String Classes** lists string and character structures and classes
- **Class Basic_string** defines facilities for character sequence manipulations.
- **Null Terminated Sequence Utilities** lists facilities for Null terminated character sequence strings.

### 6.1 Character traits

This section defines a class template `char_traits<charT>` and two specializations for `char` and `wchar_t` types.

These types are required by string and stream classes and are passed to these classes as formal parameters `charT` and `traits`.

The topics in this section are:

- **Character Trait Definitions**
- **Character Trait Requirements**
- **Character Trait Type Definitions**
- **struct char_traits<T>**
6.1.1 Character Trait Definitions

This section defines character trait definitions.

6.1.1.1 character

Any object when treated sequentially can represent text. This term is not restricted to just char and wchar_t types.

6.1.1.2 character container type

A class or type used to represent a character. This object must be POD (Plain Old Data).

6.1.1.3 traits

A class that defines types and functions necessary for handling characteristics.

6.1.1.4 NTCTS

A null character termination string is a character sequence that proceeds the null character value charT(0).

6.1.2 Character Trait Requirements

These types are required by string and stream classes and are passed to these classes as formal parameters charT and traits.

6.1.2.1 assign

Used for character type assignment.
6.1.2.2  eq

Used for bool equality checking.

    static bool eq
    (const char_type&, const char_type&);

6.1.2.3  lt

Used for bool less than checking.

    static bool lt(const char_type&, const char_type&);

6.1.2.4  compare

Used for NTCTS comparison.

    static int compare
    (const char_type*, const char_type*, size_t n);

6.1.2.5  length

Used when determining the length of a NTCTS.

    static size_t length
    (const char_type*);
Character traits

Used to find a character type in an array

```cpp
static const char_type* find
(const char_type*, int n, const char_type&);
```

### 6.1.2.7 move

Used to move one NTCTS to another even if the receiver contains the sting already.

```cpp
static char_type* move
(char_type*, const char_type*, size_t);
```

### 6.1.2.8 copy

Used for copying a NTCTS that does not contain the NTCTS already.

```cpp
static char_type* copy
(char_type*, const char_type*, size_t);
```

### 6.1.2.9 not_eof

Used for bool inequality checking.

```cpp
static int_type not_eof
(const int_type&);
```

### 6.1.2.10 to_char_type

Used to convert to a char type from an int_type

```cpp
static char_type to_char_type
(const int_type&);
```
6.1.2.11  to_int_type

Used to convert from a char type to an int_type.

\[
\text{static int_type to_int_type} \\
\text{(const char_type&);} \\
\]

6.1.2.12  eq_int_type

Used to test for equality.

\[
\text{static bool eq_int_type} \\
\text{(const int_type&, const int_type&)}; \\
\]

6.1.2.13  get_state

Used to store the state of the file buffer.

\[
\text{static state_type get_state} \\
\text{(pos_type pos);} \\
\]

6.1.2.14  eof

Used to return end of file. The value returned from eof() can be used to test against the return value of basic_istream functions such as get() to determine when another character can not be returned. It is also used to mean "not a character" on input to various functions such as basic_ostream::overflow.

\[
\text{static int_type eof();} \\
\]

6.1.3  Character Trait Type Definitions
There are several types defined in the char_traits structure for both wide and conventional char types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Defined</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>char_type</td>
<td>char values</td>
</tr>
<tr>
<td>int</td>
<td>int_type</td>
<td>integral values of char types including eof</td>
</tr>
<tr>
<td>streamoff</td>
<td>off_type</td>
<td>stream offset values</td>
</tr>
<tr>
<td>streampos</td>
<td>pos_type</td>
<td>stream position values</td>
</tr>
<tr>
<td>mbstate_t</td>
<td>state_type</td>
<td>file state values</td>
</tr>
</tbody>
</table>

### 6.1.4 struct char_traits<T>

The template structure is overloaded for both the wchar_t type `struct char_traits<wchar_t>`. This specialization is used for string and stream usage.

**NOTE**

The assign, eq and lt are the same as the =, == and < operators.

### 6.2 String Classes

The header `<string>` define string and trait classes used to manipulate character and wide character like template arguments.

### 6.3 Class basic_string

The `class basic_string` is used to store and manipulate a sequence of character like types of varying length known as strings.

Memory for a string is allocated and deallocated as necessary by member functions.

The first element of the sequence is at position zero.
The iterators used by basic_string are random iterators and as such qualifies as a reversible container.

**NOTE**

In general, the string size can be constrained by memory restrictions.

The topics in this section include:

- Constructors and Assignments
- Iterator Support
- Capacity
- Element Access
- Modifiers
- String Operations
- Non-Member Functions and Operators
- Inserters and extractors

The class basic_string can have either of two implementations:

- Refcounted.
- Non-refcounted.

The interface and functionality are identical with both implementations. The only difference is performance. Which performs best is dependent upon usage patterns in each application.

The refcounted implementation ships as the default.

**NOTE**

To enable the non-refcounted implementation un-comment
#define _EWL_NO_REFCOUNT_STRING in <ewlconfig>. The C++ library and precompiled headers must be rebuilt after making this change.

### 6.3.1 Constructors and Assignments

Constructor, destructor and assignment operators and functions.

#### 6.3.1.1 Constructors
The various basic_string constructors construct a string object for character sequence manipulations. All constructors include an Allocator argument that is used for memory allocation.

```cpp
class basic_string
```

```cpp
explicit basic_string
(const Allocator& a = Allocator());
```

This default constructor, constructs an empty string. A zero sized string that may be copied to is created.

```cpp
basic_string
(const basic_string& str,
 size_type pos = 0,
 size_type n = npos,
 const Allocator& a = Allocator());
```

This constructor takes a string class argument and creates a copy of that string, with size of the length of that string and a capacity at least as large as that string.

An exception is thrown upon failure

```cpp
basic_string
(const charT* s,
 size_type n,
 const Allocator& a = Allocator());
```

This constructor takes a const char array argument and creates a copy of that array with the size limited to the size_type argument.

The charT* argument shall not be a null pointer

An exception is thrown upon failure

```cpp
basic_string
(const charT* s,
 const Allocator& a = Allocator());
```

This constructor takes an const char array argument. The size is determined by the size of the char array.

The charT* argument shall not be a null pointer

```cpp
basic_string
(size_type n,
 charT c,
 const Allocator& a = Allocator());
```

This constructor creates a string of size_type n size repeating charT c as the filler.
A `length_error` is thrown if `n` is less than `npos`.

```cpp
template<class InputIterator>
basic_string
(InputIterator begin,
 InputIterator end,
 const Allocator& a = Allocator());
```

This iterator string takes InputIterator arguments and creates a string with its first position starts with `begin` and its ending position is `end`. Size is the distance between `beginning` and `end`.

### 6.3.1.2 Destructor

Deallocates the memory referenced by the `basic_string` object.

```cpp
~basic_string();
```

### 6.3.1.3 Assignment Operator

Assigns the input string, char array or char type to the current string.

```cpp
basic_string& operator= (const basic_string& str);
```

If `*this` and `str` are the same object has it has no effect.

```cpp
basic_string& operator=(const charT* s);
```

Used to assign a NCTCS to a string.

```cpp
basic_string& operator=(charT c);
```

Used to assign a single char type to a string.

### 6.3.1.4 Assignment & Addition Operator `basic_string`

Appends the string `rhs` to the current string.
Class `basic_string`

```cpp
string& operator+= (const string& rhs);
string& operator+= (const charT* s);
string& operator+= (charT s);
```

**Remarks**

Both of the overloaded functions construct a string object from the input `s`, and append it to the current string.

The assignment operator returns the `this` pointer.

### 6.3.2 Iterator Support

Member functions for string iterator support.

#### 6.3.2.1 begin

Returns an iterator to the first character in the string

```cpp
iterator begin();
const_iterator begin() const;
```

#### 6.3.2.2 end

Returns an iterator that is past the end value.

```cpp
iterator end();
const_iterator end() const;
```

#### 6.3.2.3 rbegin

Returns an iterator that is equivalent to

```cpp
reverse_iterator(end());
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```
6.3.2.4  rend

Returns an iterator that is equivalent to

    reverse_iterator(begin()).
    reverse_iterator rend();
    const_reverse_iterator rend() const;

6.3.3  Capacity

Member functions for determining a strings capacity.

6.3.3.1  size

Returns the size of the string.

    size_type size() const;

6.3.3.2  length

Returns the length of the string.

    size_type length() const;

6.3.3.3  max_size

Returns the maximum size of the string.

    size_type max_size() const;
6.3.3.4 resize

Resizes the string to size \( n \).

\[
\text{void resize(size_type n);} \\
\text{void resize(size_type n, charT c);} \\
\]

Remarks

If the size of the string is longer than \( \text{size_type n} \), it shortens the string to \( n \), if the size of the string is shorter than \( n \) it appends the string to size \( n \) with \( \text{charT c} \) or \( \text{charT()} \) if no filler is specified.

6.3.3.5 capacity

Returns the memory storage capacity.

\[
\text{size_type capacity() const;} \\
\]

6.3.3.6 reserve

A directive that indicates a planned change is memory size to allow for better memory management.

\[
\text{void reserve(size_type res_arg = 0);} \\
\]

6.3.3.7 clear

Erases from \( \text{begin()} \) to \( \text{end()} \).

\[
\text{void clear();} \\
\]

6.3.3.8 empty
Empties the string stored.

    bool empty() const;

Remarks

Returns true if the size is equal to zero, otherwise false.

6.3.4 Element Access

Member functions and operators for accessing individual string elements.

6.3.4.1 operator[]

An operator used to access an indexed element of the string.

    const_reference operator[](size_type pos) const;
    reference operator[](size_type pos);

6.3.4.2 at

A function used to access an indexed element of the string.

    const_reference at(size_type n) const;
    reference at(size_type n);

6.3.5 Modifiers

Operators for appending a string.

6.3.5.1 operator+=
Class basic_string

An Operator used to append to the end of a string.

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

### 6.3.5.2 append

A function used to append to the end of a string.

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos, size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
basic_string& append(InputIterator first, InputIterator last);
```

### 6.3.5.3 assign

Assigns a string, Null Terminated Character Type Sequence or char type to the string.

```
basic_string& assign(const basic_string&);
basic_string& assign(const basic_string& str, size_type pos, size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
basic_string& assign(InputIterator first, InputIterator last);
```

**Remarks**

If there is a size argument whichever is smaller the string size or argument value will be assigned.
6.3.5.4  insert

Inserts a string, Null Terminated Character Type Sequence or char type into the string.

```cpp
basic_string& insert
(size_type pos1, const basic_string& str);

basic_string& insert
(size_type pos1, const basic_string& str,
size_type pos2, size_type n);

basic_string& insert
(size_type pos, const charT* s, size_type n);

basic_string& insert(size_type pos, const charT* s);

basic_string& insert
(size_type pos, size_type n, charT c);

iterator insert(iterator p, charT c = charT());

void insert(iterator p, size_type n, charT c);

template<class InputIterator>
void insert
(iterator p, InputIterator first,
InputIterator last);
```

Remarks
May throw an exception.

6.3.5.5  erase

Erases the string

```cpp
basic_string& erase
(size_type pos = 0, size_type n = npos);

iterator erase(iterator position);

iterator erase(iterator first, iterator last);
```

Remarks
May throw an exception.
6.3.5.6 replace

Replaces the string with a string, Null Terminated Character Type Sequence or char type.

```cpp
basic_string replace pos1, size_type n1,
const basic_string& str);
```

```cpp
basic_string& replace(size_type pos1, size_type n1,
const basic_string& str, size_type pos2, size_type n2);
```

```cpp
basic_string& replace(size_type pos, size_type n1,
const charT* s, size_type n2);
```

```cpp
basic_string& replace(size_type pos, size_type n1,
charT c);
```

```cpp
basic_string& replace(iterator i1, iterator i2,
const basic_string& str);
```

```cpp
basic_string& replace(iterator i1, iterator i2,
const charT* s, size_type n);
```

```cpp
basic_string& replace(iterator i1, iterator i2, const charT* s);
```

```cpp
basic_string& replace(iterator i1, iterator i2, charT c);
```

```cpp
template<class InputIterator>
```n
```cpp
basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);
```

**Remarks**

May throw an exception,

6.3.5.7 copy

Copies a Null Terminated Character Type Sequence to a string up to the size designated.

```cpp
size_type copy(charT* s, size_type n,
size_type pos = 0) const;
```
Remarks
The function copy does not pad the string with Null characters.

6.3.5.8 swap

Swaps one string for another.

    void swap(basic_string<charT,traits,Allocator>&);

6.3.6 String Operations

Member functions for sequences of character operations.

6.3.6.1 c_str

Returns the string as a Null terminated character type sequence.

    const charT* c_str() const;

6.3.6.2 data

Returns the string as an array without a Null terminator.

    const charT* data() const;

6.3.6.3 get_allocator

Returns a copy of the allocator object used to create the string.
6.3.6.4 find

Finds a string, Null Terminated Character Type Sequence or char type in a string starting from the beginning.

```cpp
size_type find
(const basic_string& str, size_type pos = 0) const;
size_type find
(const charT* s, size_type pos, size_type n) const;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (charT c, size_type pos = 0) const;
```

Remarks
The found position or npos if not found.

6.3.6.5 rfind

Finds a string, Null Terminated Character Type Sequence or char type in a string testing backwards from the end.

```cpp
size_type rfind
(const basic_string& str, size_type pos = npos) const;
size_type rfind
(const charT* s, size_type pos, size_type n) const;
size_type rfind (const charT* s, size_type pos = npos) const;
size_type rfind(charT c, size_type pos = npos) const;
```

Remarks
The found position or npos if not found.

6.3.6.6 find_first_of
Finds the first position of one of the elements in the function's argument starting from the beginning.

```cpp
size_type find_first_of
(const basic_string& str, size_type pos = 0) const;
size_type find_first_of
(const charT* s, size_type pos, size_type n) const;
size_type find_first_of
(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const;
```

**Remarks**

The found position or `npos` if not found.

### 6.3.6.7 find_last_of

Finds the last position of one of the elements in the function's argument starting from the beginning.

```cpp
size_type find_last_of
(const basic_string& str, size_type pos = npos) const;
size_type find_last_of
(const charT* s, size_type pos, size_type n) const;
size_type find_last_of
(const charT* s, size_type pos = npos) const;
size_type find_last_of(charT c, size_type pos = npos) const;
```

**Remarks**

The found position or `npos` if not found is returned.

### 6.3.6.8 find_first_not_of

Finds the first position that is not one of the elements in the function's argument starting from the beginning.

```cpp
size_type find_first_not_of
(const basic_string& str, size_type pos = 0) const;
```
Class basic_string

size_type find_first_not_of
(const charT* s, size_type pos, size_type n) const;
size_type find_first_not_of
(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0)
const;

Remarks
The found position or npos if not found.

6.3.6.9 find_last_not_of

Finds the last position that is not one of the elements in the function's argument starting from the beginning.

size_type find_last_not_of
(const basic_string& str, size_type pos = npos) const;
size_type find_last_not_of
(const charT* s, size_type pos, size_type n) const;
size_type find_last_not_of
(const charT* s, size_type pos = npos) const;
size_type find_last_not_of(charT c, size_type pos = npos)
const;

Remarks
The found position or npos if not found.

6.3.6.10 substr

Returns a string if possible from beginning at the first arguments position to the last position.

basic_string substr
(size_type pos = 0, size_type n = npos) const;

Remarks
May throw an exception,
6.3.6.11 compare

Compares a string, substring or Null Terminated Character Type Sequence with a lexicographical comparison.

```c++
int compare(const basic_string& str) const;
int compare(
    size_type pos1, size_type n1, const basic_string& str) const;
int compare
    (size_type pos1, size_type n1,
    const basic_string& str, size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare
    (size_type pos1, size_type n1, const charT* s,
    size_type n2 = npos) const;
```

**Return**

Less than zero if the string is smaller than the argument lexicographically, zero if the string is the same size as the argument lexicographically and greater than zero if the string is larger than the argument lexicographically.

6.3.7 Non-Member Functions and Operators

Non-member functions.

6.3.7.1 operator+

Appends one string to another.

```c++
template <class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>operator+
    (const basic_string<charT,traits, Allocator>& lhs,
     const basic_string<charT,traits,Allocator>& rhs);
```
### Remarks

The combined strings are returned.

#### 6.3.7.2 operator==

Test for lexicographical equality.

```cpp
template <class charT, class traits, class Allocator>
bool operator==
(const basic_string<charT,traits,Allocator>& lhs,
 const basic_string<charT,traits,Allocator>& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
bool operator==
(const charT* lhs,const basic_string<charT,traits,Allocator>& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
bool operator==
(const basic_string<charT,traits,Allocator>& lhs,
 const charT* rhs);
```

```cpp
template<class charT, class traits, class Allocator>
bool operator==
(const basic_string<charT,traits,Allocator>& lhs, charT  rhs);
```
Return

True if the strings match otherwise false.

### 6.3.7.3 operator!=

Test for lexicographical inequality.

```cpp
template<class charT, class traits, class Allocator>
bool operator!=
(const basic_string<charT,traits,Allocator>& lhs,
 const basic_string<charT,traits,Allocator>& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
bool operator!=
(const charT* lhs,const basic_string
 <charT,traits,Allocator>& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
bool operator!=
(const basic_string<charT,traits,Allocator>& lhs,
 const charT* rhs);
```

**Remarks**

True if the strings do not match otherwise false.

### 6.3.7.4 operator<

Tests for a lexicographically less than condition.

```cpp
template <class charT, class traits, class Allocator>
bool operator<
const basic_string<charT,traits,Allocator>& lhs,
const basic_string<charT,traits,Allocator>& rhs);
```

```cpp
template <class charT, class traits, class Allocator>
bool operator<
(const charT* lhs, const basic_string
<charT,traits,Allocator>& rhs);
```
template <class charT, class traits, class Allocator>
bool operator<
    (const basic_string <charT,traits,Allocator>& lhs,
     const charT* rhs);

Remarks

Returns \textit{true} if the first argument is lexicographically less than the second argument otherwise \textit{false}.

6.3.7.5 \textit{operator>}

Tests for a lexicographically greater than condition.

template <class charT, class traits, class Allocator>
bool operator>
    (const basic_string <charT,traits,Allocator>& lhs,
     const basic_string <charT,traits,Allocator>& rhs);

template <class charT, class traits, class Allocator>
bool operator>
    (const charT* lhs,const basic_string
     <charT,traits,Allocator>& rhs);

template <class charT, class traits, class Allocator>
bool operator>
    (const basic_string <charT,traits,Allocator>& lhs,
     const charT* rhs);

Remarks

Returns \textit{true} if the first argument is lexicographically greater than the second argument otherwise \textit{false}.

6.3.7.6 \textit{operator<=}

Tests for a lexicographically less than or equal to condition.

template <class charT, class traits, class Allocator>
bool operator<=
(const basic_string & lhs,
const basic_string & rhs);
template <class charT, class traits, class Allocator>
bool operator<=
(const charT* lhs,
const basic_string & rhs);
template <class charT, class traits, class Allocator>
bool operator<=
(const basic_string & lhs, const charT* rhs);

Remarks

Returns true if the first argument is lexicographically less than or equal to the second argument otherwise false.

6.3.7.7 operator>=

Tests for a lexicographically greater than or equal to condition.

template <class charT, class traits, class Allocator>
bool operator>=
(const basic_string & lhs,
const basic_string & rhs);
template <class charT, class traits, class Allocator>
bool operator>=
(const charT* lhs,
const basic_string & rhs);
template <class charT, class traits, class Allocator>
bool operator>=
(const basic_string & lhs, const charT* rhs);

Remarks

Returns true if the first argument is lexicographically greater than or equal to the second argument otherwise false.
6.3.7.8 swap

This non member \texttt{swap} exchanges the first and second arguments.

\begin{verbatim}
template <class charT, class traits, class Allocator>
void swap
(basic_string<charT,traits,Allocator>& lhs,
basic_string <charT,traits,Allocator>& rhs);
\end{verbatim}

6.3.8 Inserters and extractors

Overloaded inserters and extractors for basic\_string types.

6.3.8.1 \texttt{operator\textgreater\textgreater}

Overloaded \texttt{extractor} for stream input operations.

\begin{verbatim}
template <class charT, class traits, class Allocator>
basic_istream<charT,traits>& operator\textgreater\textgreater
(basic_istream<charT,traits>& is,
basic_string<charT,traits,Allocator>& str);
\end{verbatim}

Remarks

Characters are extracted and appended until \texttt{n} characters are stored or \texttt{end-of-file} occurs on the input sequence;

6.3.8.2 \texttt{operator\textless\textless}

Inserts characters from a string object from into a output stream.

\begin{verbatim}
template <class charT, class traits, class Allocator>
basic_ostream<charT, traits>& operator\textless\textless
(basic_ostream<charT, traits>& os,
const basic_string <charT,traits,Allocator>& str);
\end{verbatim}
6.3.8.3 getline

Extracts characters from a stream and appends them to a string.

```
template <class charT, class traits, class Allocator>
basic_istream<charT,traits>& getline
(basic_istream<charT,traits>& is,
basic_string<charT,traits,Allocator>& str,charT delim);
template <class charT, class traits, class Allocator>
basic_istream<charT,traits>& getline
(basic_istream<charT,traits>& is,
basic_string<charT,traits,Allocator>& str)
```

Remarks

Extracts characters from a stream and appends them to the string until the end-of-file occurs on the input sequence (in which case, the getline function calls setstate(eofbit) or the delimiter is encountered in which case, the delimiter is extracted but not appended.

If the function extracts no characters, it calls setstate(failbit) in which case it may throw an exception.

6.4 Null Terminated Sequence Utilities

The standard requires C++ versions of the standard libraries for use with characters and Null Terminated Character Type Sequences.

6.4.1 Character Support

The standard provides for namespace and character type support.

```
<table>
<thead>
<tr>
<th>&lt;cctype.h&gt;</th>
<th>&lt;cwctype.h&gt;</th>
<th>&lt;cwctype.h&gt;</th>
<th>&lt;cwctype.h&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum</td>
<td>iswalnum</td>
<td>isprint</td>
<td>iswprint</td>
</tr>
</tbody>
</table>
```

*Table 6-2. Character support testing*

*Table continues on the next page...*
### 6.4.2 String Support

The standard provides for namespace and wide character type for Null Terminated Character Type Sequence functionality.

#### Table 6-3. String support testing

<table>
<thead>
<tr>
<th><code>&lt;cstring.h&gt;</code></th>
<th><code>&lt;wchar.h&gt;</code></th>
<th><code>&lt;cstring.h&gt;</code></th>
<th><code>&lt;wchar.h&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>memchr</td>
<td>wmemchr</td>
<td>strerror</td>
<td></td>
</tr>
<tr>
<td>memcmp</td>
<td>wmemcmp</td>
<td>strlen</td>
<td>wcslen</td>
</tr>
<tr>
<td>memcpy</td>
<td>wmemcpy</td>
<td>strncmp</td>
<td>wcsncmp</td>
</tr>
<tr>
<td>memmove</td>
<td>wmemmove</td>
<td>strncpy</td>
<td>wcsncpy</td>
</tr>
<tr>
<td>memset</td>
<td>wmemset</td>
<td>strncpy</td>
<td>wcsncpy</td>
</tr>
<tr>
<td>strcat</td>
<td>wcscat</td>
<td>strpbrk</td>
<td>wcssbrk</td>
</tr>
<tr>
<td>strchr</td>
<td>wcscchr</td>
<td>strchr</td>
<td>wcschr</td>
</tr>
<tr>
<td>strcmp</td>
<td>wcscmp</td>
<td>strspn</td>
<td>wcsspn</td>
</tr>
<tr>
<td>strcoll</td>
<td>wcscoll</td>
<td>strstr</td>
<td>wcsstr</td>
</tr>
<tr>
<td>strcpy</td>
<td>wcscpy</td>
<td>strtok</td>
<td>wcstok</td>
</tr>
<tr>
<td>strcspn</td>
<td>wcscspn</td>
<td>strxfrm</td>
<td>wcsxfrm</td>
</tr>
<tr>
<td>mbstate_t</td>
<td>size_t</td>
<td>wint_t</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td>WCHAR_MAX</td>
<td>WCHAR_MIN</td>
</tr>
</tbody>
</table>

### 6.4.3 Input and Output Manipulations
The standard provides for namespace and wide character support for manipulation and conversions of input and output and character and character sequences.

### Table 6-4. Additional `<wchar.h>` and `<stdlib.h>` support

<table>
<thead>
<tr>
<th><code>&lt;wchar.h&gt;</code></th>
<th><code>&lt;wchar.h&gt;</code></th>
<th><code>&lt;wchar.h&gt;</code></th>
<th><code>&lt;&lt;stdlib.h&gt;&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>btowc</code></td>
<td><code>mbtowc</code></td>
<td><code>wctomb</code></td>
<td><code>atol</code></td>
</tr>
<tr>
<td><code>fgetwc</code></td>
<td><code>mbsinit</code></td>
<td><code>wcscoll</code></td>
<td><code>atof</code></td>
</tr>
<tr>
<td><code>fgetws</code></td>
<td><code>mbstowcs</code></td>
<td><code>wcsftime</code></td>
<td><code>atoi</code></td>
</tr>
<tr>
<td><code>fputwc</code></td>
<td><code>putwc</code></td>
<td><code>wctype</code></td>
<td><code>mblen</code></td>
</tr>
<tr>
<td><code>fputws</code></td>
<td><code>putwchar</code></td>
<td><code>wcstol</code></td>
<td><code>mbstowcs</code></td>
</tr>
<tr>
<td><code>fwrite</code></td>
<td><code>swscanf</code></td>
<td><code>wcsrtombs</code></td>
<td><code>mbtowc</code></td>
</tr>
<tr>
<td><code>fwprintf</code></td>
<td><code>swprintf</code></td>
<td><code>wcstoul</code></td>
<td><code>strtod</code></td>
</tr>
<tr>
<td><code>fscanf</code></td>
<td><code>ungetwc</code></td>
<td><code>wctob</code></td>
<td><code>strtol</code></td>
</tr>
<tr>
<td><code>getwc</code></td>
<td><code>vfprintf</code></td>
<td><code>wprintf</code></td>
<td><code>strtoul</code></td>
</tr>
<tr>
<td><code>getwchar</code></td>
<td><code>vwprintf</code></td>
<td><code>wscanf</code></td>
<td><code>wctomb</code></td>
</tr>
<tr>
<td><code>mbrlen</code></td>
<td><code>vswprintf</code></td>
<td></td>
<td><code>wcsrtombs</code></td>
</tr>
</tbody>
</table>
Chapter 7
Localization Library

This chapter describes components that the C++ library that may use for porting to
different cultures.

Much of named locales is implementation defined behavior and is not portable between
vendors. This document specifies the behavior of EWL C++. Other vendors may not
provide this functionality, or may provide it in a different manner.

This chapter is constructed in the following sub sections and uses the ISO (International
Organization for Standardization) C++ Standard as a guide:

- Supported Locale Names
- Strings and Characters in Locale Data Files
- Locales
- Standard Locale Categories
- C Library Locales

7.1 Supported Locale Names

EWL C++ predefines only two names: "c" and "".

However, other names sent to the locale constructor are interpreted as file names
containing data to create a named locale. So localizing your program is as easy as
creating a data file specifying the desired behavior. The format for this data file is
outlined below for each different facet.

A locale is a collection of facets. And a facet is a class that provides a certain behavior.
The "c" locale contains the following facets:

- ctype<char> & ctype<wchar_t>
- codecvt<char, char, mbstate_t> & codecvt<wchar_t, char, mbstate_t>
- num_get<char> & num_get<wchar_t>
Strings and Characters in Locale Data Files

- `num_put<char>` & `num_put<wchar_t>`
- `numpunct<char>` & `numpunct<wchar_t>`
- `collate<char>` & `collate<wchar_t>`
- `time_get<char>` & `time_get<wchar_t>`
- `time_put<char>` & `time_put<wchar_t>`
- `money_get<char>` & `money_get<wchar_t>`
- `money_put<char>` & `money_put<wchar_t>`
- `moneypunct<char, bool>` & `moneypunct<wchar_t, bool>`
- `messages<char>` & `messages<wchar_t>`

A named locale replaces many of these facets with "_byname" versions, whose behavior can vary based on the name passed.

- `ctype_byname<char>` & `ctype_byname<wchar_t>`
- `codecvt_byname<char, char, mbstate_t>` & `codecvt_byname<wchar_t, char, mbstate_t>`
- `numpunct_byname<char>` & `numpunct_byname<wchar_t>`
- `collate_byname<char>` & `collate_byname<wchar_t>`
- `time_get_byname<char>` & `time_get_byname<wchar_t>`
- `time_put_byname<char>` & `time_put_byname<wchar_t>`
- `moneypunct_byname<char, bool>` & `moneypunct_byname<wchar_t, bool>`
- `messages_byname<char>` & `messages_byname<wchar_t>`

The behavior of each of these "_byname" facets can be specified with a data file. A single data file can contain data for all of the byname facets. That way, when you code:

```cpp
locale myloc("MyLocale");
```

then the file "MyLocale" will be used for each "_byname" facet in myloc.

**NOTE**

Unnamed namespaces are displayed using a compiler generated unique name that has the form: `__unnamed_<filename>` where `<filename>` is the source file name of the main translation unit that contains the unnamed namespace.

7.2 Strings and Characters in Locale Data Files

The named locale facility involves reading strings and characters from files. This document gives the details of the syntax used to enter strings and characters.
7.2.1 Character Syntax

Characters in a locale data file can in general appear quoted (') or not. For example:

```plaintext
thousands_sep = ,
thousands_sep = ','
```

Both of the above statements set thousands_sep to a comma. Quotes might be necessary
to disambiguate the intended character from ordinary whitespace. For example, to set the
thousands_sep to a space character, quotes must be used:

```plaintext
thousands_sep = ' '
```

The whitespace appearing before and after the equal sign is not necessary and
insignificant.

7.2.2 Escape sequences

The usual C escape sequences are recognized. For example, to set the thousands_sep to
the single quote character, an escape sequence must be used:

```plaintext
thousands_sep = \'
```

The recognized escape sequences are:

- \n - newline
- \t - horizontal tab
- \v - vertical tab
- \b - backspace
- \r - carriage return
- \f - form feed
- \a - alert
- \\ - \n
- \? - ?
- \" - "
- \' - 
- \u \U - universal character
- \x - hexadecimal character
- \000 - octal character
The octal character may have from 1 to 3 octal digits (digits must be in the range [0, 7]). The parser will read as many digits as it can to interpret a valid octal number. For example:

\18

This is the character '\1' followed by the character '8'.

\17

But this is the single character '\17'.

The hexadecimal and universal character formats are all identical with each other, and have slightly relaxed syntax compared to the formats specified in the standard. The x (or u or U) is followed by zero to sizeof(charT)*CHAR_BIT/4 hexadecimal digits. charT is char when reading narrow data, and wchar_t when reading wide data (even when reading wide data from a narrow file). On Macintosh and Windows this translates to 0 to 2 digits when reading a char, and from 0 to 4 digits when reading a wchar_t. Parsing the character is terminated when either the digit limit has been reached, or a non-hexadecimal digit has been reached. If there are 0 valid digits, then the character is read as '\0'. Example (assume a 8 bit char and 16 bit wchar_t):

\x01234

When reading narrow data this is the following sequence of 4 char's: '\1' '2' '3' '4'

The '\x01' is read as one character, but the following '2' is not included because a 8 bit char can only hold 2 hex digits.

When reading wide data the above example parses to the following two wchar_t's:

L'\x123' L'4'

The '\x0123' is read as one wchar_t, but the following '4' is not included because a 16 bit wchar_t can only hold 4 hex digits.

7.2.3 Errors

If a character is expected, but an end of file occurs, then failbit is set. If a character is started with a single quote, and end of file occurs before the character within the quotes can be read, or if a closing quote is not found directly after the character, then failbit will be set. Depending on the context of when the character is being read, setting failbit may or may not cause a runtime error to be thrown.
7.2.4 String Syntax

Strings can be quoted or not (using "). If the string contains white space, then it must be quoted. For example:

Hi there!

This would be parsed as two strings: "Hi" and "there!". But the following is one string:

"Hi there!"

If a string begins with quotes, but does not end with a quote (before end of file), then failbit will be set. This may nor may not cause a runtime error to be thrown (depending on the context).

Any of the escape sequences described under character syntax are allowed within strings. But within strings, single quotes do not delimit characters. Instead single quotes are just another character in the string. Note that you can use \" to place the string quote character within a string.

7.3 Locales

The header <locale> defines classes used to contain and manipulate information for a locale.

- Class locale
- Locale Types
- Locale Members
- Locale Operators
- Locale Static Members
- Locale Globals
- Convenience Interfaces
- Character Classification
- Character Conversions

7.3.1 Class locale
The class locale contains a set of facets for locale implementation. These facets are as if they were and index and an interface at the same time.

### 7.3.1.1 Combined Locale Names

Two locale constructors can result in a new locale whose name is a combination of the names of two other locales:

```
locale(const locale& other, const char* std_name, category);
locale(const locale& other, const locale& one, category);
```

If other has a name (and if one has a name in the case of the second constructor), then the resulting locale's name is composed from the two locales' names. A combined name locale has the format:

```
collate_name/ctype_name/monetary_name/numeric_name/time_name/messages_name
```

Each name is the name of a locale from which that category of facets was copied.

The locale loc is created from two locales: other and one. The facets in the categories collate and numeric are taken from one. The rest of the facets are taken from other. The name of the resulting locale is:

```
one/other/other/one/other/other
```

The locale loc2 is created from the "C" locale and from loc (which already has a combined name). It takes only the monetary and collate facets from loc, and the rest from "C":

```
one/C/other/C/C/C
```

Using this format, two locales can be compared by name, and if their names are equal, then they have the same facets.

**Listing: Locale example usage:**

```c++
#include <locale>
#include <iostream>

int main()
{
    using std::locale;
    locale loc(locale("other"), locale("one"),
              locale::collate | locale::numeric);
```

Locales

EWL C++ Library Reference Manual, Rev. 10.x, 02/2014

Freescale Semiconductor, Inc.
std::cout << loc.name() << '\n';
locale loc2(locale(), loc, locale::monetary |
    locale::collate);
std::cout << loc2.name() << '\n';
}

7.3.2 Locale Types

This library contains various types specific for locale implementation.

7.3.2.1 locale::Category

An integral type used as a mask for all types.

typedef int category;

Each locale member function takes a locale::category argument based on a corresponding facet.

### Table 7-1. Locale Category Facets

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes Facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td>collate&lt;char&gt;, collate&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>ctype</td>
<td>ctype&lt;char&gt;, ctype&lt;wchar_t&gt;, codecvt&lt;char,char,mbstate_t&gt;, codecvt&lt;wchar_t,char,mbstate_t&gt;</td>
</tr>
<tr>
<td>messages</td>
<td>messages&lt;char&gt;, messages&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>monetary</td>
<td>money_punct&lt;char&gt;, money_punct&lt;wchar_t&gt;, money_punct&lt;char,true&gt;, money_punct&lt;wchar_t,true&gt;, money_get&lt;char&gt;, money_get&lt;wchar_t&gt;, money_put&lt;char&gt;, money_put&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>numeric</td>
<td>num_punct&lt;char&gt;, num_punct&lt;wchar_t&gt;, num_get&lt;char&gt;, num_get&lt;wchar_t&gt;, num_put&lt;char&gt;, num_put&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>time</td>
<td>time_get&lt;char&gt;, time_get&lt;wchar_t&gt;, time_put&lt;char&gt;, time_put&lt;wchar_t&gt;</td>
</tr>
</tbody>
</table>
An implementation is included for each `facet` template member of a `category`.

### Table 7-2. Required Instantiations

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes Facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td><code>collatebyname&lt;char&gt;, collatebyname&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>ctype</td>
<td><code>ctypebyname&lt;char&gt;, ctypebyname&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>messages</td>
<td><code>messagesbyname&lt;char&gt;, messagesbyname&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>numeric</td>
<td><code>numpunctbyname&lt;char&gt;, numpunctbyname&lt;wchar_t&gt;, num_get&lt;C, InputIterator&gt;, num_put&lt;C, OutputIterator&gt;</code></td>
</tr>
<tr>
<td>time</td>
<td><code>time_get&lt;char, InputIterator&gt;, time_getbyname&lt;char, InputIterator&gt;, time_get&lt;wchar_t, OutputIterator&gt;, time_getbyname&lt;wchar_t, OutputIterator&gt;, time_put&lt;char, OutputIterator&gt;, time_putbyname&lt;char, OutputIterator&gt;, time_put&lt;wchar_t, OutputIterator&gt;, time_putbyname&lt;wchar_t, OutputIterator&gt;</code></td>
</tr>
</tbody>
</table>

7.3.2.2 `locale::facet`

The class `facet` is the base class for `locale` feature sets.

**Listing: class locale::facet synopsis**

```cpp
namespace std {
    class locale::facet {
        protected:
        explicit facet(size_t refs = 0);
        virtual ~facet();
    private:
        facet(const facet&); // not defined
        void operator=(const facet&); // not defined
    }
}
```

7.3.2.3 `locale::id`

The class `locale::id` is used for an index for locale facet identification.
Listing: class locale::id synopsis

namespace std {
class locale::id {
public:
id();
private:
void operator=(const id&); // not defined
id(const id&); // not defined 
}

7.3.2.4 Constructors

Constructs an object of locale.

locale() throw();
locale(const locale& other) throw();
explicit locale(const char* std_name);
locale(const locale& other, const char* std_name, category);
template <class Facet> locale(const locale& other, Facet* f);
locale(const locale& other, const locale& one, category cats);

Remarks

std::locale a_locale(""); is an example use of the constructor: explicit locale(const char* std_name);. The "" locale will attempt to read the environment variable EWL_DEFAULT_LOCALE and create a locale with the associated string. If getenv("EWL_DEFAULT_LOCALE") returns null, then "C" is used. There is no data file associated with the "C" locale. The "C" locale is coded directly into EWL C++.

7.3.2.5 destructor

Removes a locale object.

~locale() throw();
7.3.3 Locale Members

Member functions of the class `locale`.

7.3.3.1 combine

Creates a copy of the `locale` except for the type `Facet` of the argument.

```cpp
template <class Facet> locale combine(const locale& other);
```

Remarks
The newly created locale is returned.

7.3.3.2 name

Returns the name of the `locale`.

```cpp
basic_string<char> name() const;
```

Remarks
Returns the name of the locale or "*" if there is none.

7.3.4 Locale Operators

The class `locale` has overloaded operators.

7.3.4.1 operator ==

The locale equality operator.
bool operator==(const locale& other) const;

Remarks
The equality operator returns true if both arguments are the same locale.

7.3.4.2 operator !=

The locale non-equality operator

bool operator!=(const locale& other) const;

Remarks
The non-equality operator returns true if the locales are not the same.

7.3.4.3 operator ()

Compares two strings using use_facet<collate<> >.

template <class charT,  
class Traits, class Allocator>
bool operator()(
    const basic_string<charT,Traits,Allocator>& s1,  
    const basic_string<charT,Traits,Allocator>& s2)
const;

Remarks
Returns true if the first argument is less than the second argument for ordering.

7.3.5 Locale Static Members

This section describes local static members.
### 7.3.5.1 global

Installs a new global locale.

```cpp
static locale global(const locale& loc);
```

**Remarks**

Global returns the previous locale.

### 7.3.5.2 classic

Sets the locale to "C" locale equivalent to `locale("C")`.

```cpp
static const locale& classic();
```

**Remarks**

This function returns the "C" locale.

### 7.3.6 Locale Globals

Locale has two global functions.

#### 7.3.6.1 use_facet

Retrieves a reference to a facet of a locale.

```cpp
template <class Facet> const Facet& use_facet
    (const locale& loc);
```

**Remarks**

Throws a `bad_cast` exception if `has_facet` is false.

The function returns a facet reference to corresponding to its argument.
7.3.6.2 has_facet

Tests a locale to see if a facet is present

```cpp
template <class Facet> bool has_facet
(const locale& loc) throw();
```

Remarks
If a facet requested is present has_facet returns true.

7.3.7 Convenience Interfaces

Character classification functionality is provided for in the locale class.

7.3.8 Character Classification

In the character classification functions true is returned if the function evaluates to true.

Listing: Character Classification

```cpp
template <class charT> bool isspace (charT c, const locale& loc);
template <class charT> bool isprint (charT c, const locale& loc);
template <class charT> bool iscntrl (charT c, const locale& loc);
template <class charT> bool isupper (charT c, const locale& loc);
template <class charT> bool islower (charT c, const locale& loc);
template <class charT> bool isalpha (charT c, const locale& loc);
template <class charT> bool isdigit (charT c, const locale& loc);
template <class charT> bool ispunct (charT c, const locale& loc);
template <class charT> bool isxdigit(charT c, const locale& loc);
template <class charT> bool isalnum (charT c, const locale& loc);
template <class charT> bool isgraph (charT c, const locale& loc);
```
7.3.9  Character Conversions

Character conversion functionality is provided for in the locale class.

7.3.9.1  toupper

Converts to upper case character using the locale specified.

    template <class charT> charT toupper
    (charT c, const locale& loc) const;

Remarks

Returns the upper case character.

7.3.9.2 tolower

Converts to a lower case character using the locale specified.

    template <class charT> charT tolower
    (charT c, const locale& loc) const;

Remarks

Returns the lower case character.

7.4  Standard Locale Categories

The standard provides for various locale categories for providing formatting and manipulation of data and streams.

- The Ctype Category
- The Numeric Category
- The Collate Category
- The Time Category
7.4.1 The Ctype Category

The type `ctype_base` provides for const enumerations.

Listing: Ctype Category

```cpp
namespace std {
    class ctype_base
    {
    public:
        enum mask
        {
            alpha  = 0x0001,
            blank  = 0x0002,
            cntrl  = 0x0004,
            digit  = 0x0008,
            graph  = 0x0010,
            lower  = 0x0020,
            print  = 0x0040,
            punct  = 0x0080,
            space  = 0x0100,
            upper  = 0x0200,
            xdigit = 0x0400,
            alnum  = alpha | digit
        };
    };}
```

7.4.1.1 Template Class Ctype

The class `ctype` provides for character classifications.
7.4.1.1.1  is

An overloaded function that tests for or places a mask.

```cpp
bool is(mask m, charT c) const;
```

Test if `c` matches the mask `m`.

Returns true if the char `c` matches mask.

```cpp
const charT* is(const charT* low, const charT* high,
                 mask* vec) const;
```

Fills between the low and high with the mask argument.

Returns the second argument.

7.4.1.1.2  scan_is

Scans the range for a mask value.

```cpp
const charT* scan_is(mask m, const charT* low, const charT* high) const;
```

Remarks

Returns a pointer to the first character in the range that matches the mask, or the `high` argument if there is no match.

7.4.1.1.3  scan_not

Scans the range for exclusion of the mask value.

```cpp
const charT* scan_not(mask m, const charT* low, const charT* high) const;
```

Remarks

Returns a pointer to the first character in the range that does not match the mask, or the `high` argument if all characters match.
7.4.1.1.4  toupper

Converts to a character or a range of characters to uppercase.

```
charT toupper(charT) const;
const charT* toupper (charT* low, const charT* high) const;
```

Remarks

Returns the converted char if it exists.

7.4.1.1.5  tolower

Converts to a character or a range of characters to lowercase.

```
charT tolower(charT c) const;
const charT* tolower(charT* low, const charT* high) const;
```

Remarks

Returns the converted char if it exists.

7.4.1.1.6  widen

Converts a char or range of char type to the charT type.

```
charT widen(char c) const;
const char* widen (const char* low, const char* high, charT* to) const;
```

Remarks

The converted charT is returned.

7.4.1.1.7  narrow

Converts a charT or range of charT type to the char type.

```
char narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT*, char dfault, char* to) const;
```
Remarks
The converted `char` is returned.

7.4.1.2 ctype Virtual Functions

Virtual functions must be overloaded in the locale.

7.4.1.2.1 do_is

Implements the function `is`.

```cpp
bool do_is (mask m, charT c) const;
const charT* do_is
  (const charT* low, const charT* high, mask* vec) const;
```

7.4.1.2.2 do_scan_is

Implements the function `scan_is`.

```cpp
const charT* do_scan_is(mask m,
  const charT* low, const charT* high) const;
```

7.4.1.2.3 do_scan_not

Implements the function `scan_not`.

```cpp
const charT* do_scan_not(mask m,
  const charT* low, const charT* high) const;
```

7.4.1.2.4 do_toupper

Implements the function `toupper`.

```cpp
charT do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;
```
7.4.1.2.5 do_tolower

Implements the function tolower.

```cpp
charT do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;
```

7.4.1.2.6 do_widen

Implements the function widen.

```cpp
charT do_widen(char c) const;
const char* do_widen(const char* low, const char* high, charT* dest) const;
```

7.4.1.2.7 do_narrow

Implements the function narrow.

```cpp
char do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;
```

7.4.1.3 Template class ctype_byname

The template class ctype_byname has several responsibilities.

- character classification
- conversion to upper/lower case
- conversion to/from char

7.4.1.3.1 ctype_byname Constructor

```cpp
explicit ctype_byname(const char*, size_t refs = 0);
```
The facet ctype has several responsibilities:

- character classification
- conversion to upper/lower case
- conversion to/from char

The first two of these items can be customized with \texttt{ctype\_byname}. If you construct \texttt{ctype\_byname} with a const char* that refers to a file, then that file is scanned by \texttt{ctype\_byname}'s constructor for information to customize character classification, and case transformation tables.

\begin{verbatim}
ctype\_byname\texttt{<char>} ct("en\_US");
// looks for the file "en\_US"
\end{verbatim}

If the file "en\_US" exists, has ctype data in it, and there are no syntax errors in the data, then \texttt{ct} will behave as dictated by that data. If the file exists, but does not have ctype data in it, then the facet will behave as if it were constructed with "C". If the file has ctype data in it, but there is a syntax error in the data, or if the file does not exist, then a \texttt{std::runtime\_error} is thrown.

For \texttt{ctype\_byname\texttt{<char>}}, the ctype data section begins with:

\begin{verbatim}
$\texttt{ctype\_narrow}
\end{verbatim}

For \texttt{ctype\_byname\texttt{<wchar\_t>}}, the ctype data section begins with:

\begin{verbatim}
$\texttt{ctype\_wide}
\end{verbatim}

\subsection*{7.4.1.3.2 Classification}

The classification table is created with one or more entries of the form:

\begin{verbatim}
ctype[character1 - character2] =
  ctype\_classification | ...
ctype[character] = ctype\_classification | ...
\end{verbatim}

where character, character1 and character2 are characters represented according to the rules for \textit{Strings and Characters in Locale Data Files}. The characters may appear as normal characters:
ctype[a - z]
cctype[\a' - 'z']

or as octal, hexadecimal or universal:

cctype[\101]
cctype[\x41]
cctype[\u41]

The usual escape sequences are also recognized: \n, \t, \a, \\, \' and so on.

On the right hand side of the equal sign, ctype_classification is one of:

- alpha
- blank
- cntrl
- digit
- graph
- lower
- print
- punct
- space
- upper
- xdigit

An | can be used to assign a character, or range of characters, more than one classification. These keywords correspond to the names of the enum ctype_base::mask, except that alnum is not present. To get alnum simply specify "alpha | digit". The keyword blank is introduced, motivated by C99's isblank function.

Each of these keywords represent one bit in the ctype_base::mask. Thus for each entry into the ctype table, one must specify all attributes that apply. For example, in the "C" locale a-z are represented as:

cctype[\a' - 'z'] =

\xdigit | lower | alpha | graph | print

7.4.1.3.3 Case Transformation
Case transformation is usually handled by a table that maps each character to itself, except for those characters being transformed - which are mapped to their transformed counterpart. For example, a lower case map might look like:

```c
lower['a'] == 'a'
lower['A'] == 'a'
```

This is represented in the ctype data as two tables: lower and upper. You can start a map by first specifying that all characters map to themselves:

```c
lower['\0' - '\xFF'] = '\0' - '\xFF'
```

You can then override a subrange in this table to specify that 'A' - 'Z' maps to 'a' - 'z':

```c
lower['A' - 'Z'] = 'a' - 'z'
```

These two statements have completely specified the lower case mapping for an 8 bit char. The upper case table is similar. For example, here is the specification for upper case mapping of a 16 bit wchar_t in the "C" locale:

```c
upper['\0' - '\xFFFF'] = '\0' - '\xFFFF'
upper['a' - 'z'] = 'A' - 'Z'
```

Below is the complete "C" locale specification for both ctype_byname<char> and ctype_byname<wchar_t>. Note that a "C" data file does not actually exist. But if you provided a locale data file with this information in it, then the behavior would be the same as the "C" locale.

**Listing: Example of "C" Locale**

```c
$ctype_narrow
ctype['\x00' - '\x08'] = cntrl
ctype['\x09'] = cntrl | space | blank
ctype['\x0A' - '\x0D'] = cntrl | space
ctype['\x0E' - '\x1F'] = cntrl
ctype['\x20'] = space | blank | print
ctype['\x21' - '\x2F'] = punct | graph | print
ctype['\x30' - '\x39'] = digit | xdigit | graph | print
ctype['\x3A' - '\x40'] = punct | graph | print
ctype['\x41' - '\x46'] = xdigit | upper | alpha | graph | print
ctype['\x47' - '\x5A'] = upper | alpha | graph | print
```
ctype['\x5B' - '\x60'] = punct | graph | print
ctype['\x61' - '\x66'] = xdigit | lower | alpha | graph | print
ctype['\x67' - '\x7A'] = lower | alpha | graph | print
ctype['\x7B' - '\x7E'] = punct | graph | print
cctype['\x7F']          = cntrl

lower['\0' - '\xFF'] = '\0' - '\xFF'
lower['A' - 'Z'] = 'a' - 'z'

upper['\0' - '\xFF'] = '\0' - '\xFF'
upper['a' - 'z'] = 'A' - 'Z'

$ctype_wide
ctype['\x00' - '\x08'] = cntrl
ctype['\x09']          = cntrl | space | blank
ctype['\x0A' - '\x0D'] = cntrl | space
ctype['\x0E' - '\x1F'] = cntrl
ctype['\x20']          = space | blank | print
ctype['\x21' - '\x2F'] = punct | graph | print
ctype['\x30' - '\x39'] = digit | xdigit | graph | print
ctype['\x3A' - '\x40'] = punct | graph | print
cctype['\x41' - '\x46'] = xdigit | upper | alpha | graph | print
cctype['\x47' - '\x5A'] = upper | alpha | graph | print
cctype['\x5B' - '\x60'] = punct | graph | print
cctype['\x61' - '\x66'] = xdigit | lower | alpha | graph | print
cctype['\x67' - '\x7A'] = lower | alpha | graph | print
cctype['\x7B' - '\x7E'] = punct | graph | print
cctype['\x7F']          = cntrl

lower['\0' - '\xFFFF'] = '\0' - '\xFFFF'
lower['A' - 'Z'] = 'a' - 'z'

upper['\0' - '\xFFFF'] = '\0' - '\xFFFF'
upper['a' - 'z'] = 'A' - 'Z'
7.4.1.4 ctype Specializations

The category ctype has various specializations to help localization.

The class `ctype<char>` has four protected data members:

- `const mask* __table_;`
- `const unsigned char* __lower_map_;`
- `const unsigned char* __upper_map_;`
- `bool __owns_;`

Each of the pointers refers to an array of length `ctype<char>::table_size`. The destructor `~ctype<char>()` will delete `__table_` if `__owns_` is true, but it will not delete `__lower_map_` and `__upper_map_`. The derived class destructor must take care of deleting these pointers if they are allocated on the heap (`ctype<char>` will not allocate these pointers). A derived class can set these pointers however it sees fit, and have `ctype<char>` implement all of the rest of the functionality.

The class `ctype<wchar_t>` has three protected data members:

```
Metrowerks::range_map<charT, ctype_base::mask> __table_;
```

```
Metrowerks::range_map<charT, charT> __lower_map_;
```

```
Metrowerks::range_map<charT, charT> __upper_map_;
```

The class `range_map` works much like the tables in `ctype<char>` except that they are sparse tables. This avoids having tables of length 0xFFFF. These tables map the first template parameter into the second.

**Listing: The range_map interface**

```
template <class T, class U>
class range_map
{
public:
    U operator[](const T& x) const;
    void insert(const T& x1, const T& x2, const U& y1, const U& y2);
```
When constructed, the range_map implicitly holds a map of all T that map to U(). Use of the insert methods allows exceptions to that default mapping. For example, the first insert method maps the range [x1 - x2] into [y1 - y2]. The second insert method maps the x-range into a constant: y1. And the third insert method maps the single T(x1) into U(y1). The method clear() brings the range_map back to the default setting: all T map into U().

A class derived from ctype<wchar_t> can fill __table__, __lower_map_ and __upper_map_ as it sees fit, and allow the base class to query these tables. For an example see ctype_byname<wchar_t>.

7.4.1.5 Specialized Ctype Constructor and Destructor

Specialized ctype<char> and ctype<wchar_t> constructors and destructors.

7.4.1.5.1 Constructor

Constructs a ctype object.

```cpp
explicit ctype
(const mask* tbl = 0, bool del = false,
 size_t refs = 0);
```

7.4.1.5.2 destructor

Removes a ctype object.

```cpp
-ctype();
```

7.4.1.5.3 Specialized Ctype Members

Listing: Several Ctype members are specialized in the standard library

Specialized
ctype<char> and
ctype<wchar_t> member functions.
7.4.1.6  ctype<Char> Static Members

Specialized ctype<char> static members are provided.

7.4.1.6.1  classic_table

Determines the classification of characters in the "C" locale.

    static const mask* classic_table() throw();

Remarks

Returns to a table that represents the classification in a "C" locale.
7.4.1.7 Class ctype_byname<char>

A specialization of ctype_byname of type char.

7.4.1.7.1 ctype_byname<char> Constructor

```cpp
explicit ctype_byname(const char*, size_t refs = 0);
```

The facet ctype has several responsibilities:

• character classification
• conversion to upper/lower case
• conversion to/from char

For a full and complete description of this facet specialization see Ctype_byname Constructor which list the process in greater detail.

7.4.1.8 Template Class Codecvt

A class used for converting one character encoded types to another. For example, from wide character to multibyte character sets.

7.4.1.8.1 codecvt Members

Member functions of the codecvt class.

7.4.1.8.1.1 out

Convert internal representation to external.

```cpp
result out(
    stateT& state, const internT* from,
    const internT* from_end, const internT* & from_next,
    externT* to, externT* to_limit,
    externT* & to_next) const;
```
7.4.1.8.1.2  unshift

Converts the shift state.

```cpp
result unshift(stateT& state,
       externT* to, externT* to_limit, externT*& to_next) const;
```

7.4.1.8.1.3  in

Converts external representation to internal.

```cpp
result in(stateT& state, const externT* from,
       const externT* from_end, const externT*&
       from_next, internT* to, internT* to_limit,
       internT*& to_next) const;
```

7.4.1.8.1.4  always_noconv

Determines if no conversion is ever done.

```cpp
bool always_noconv() const throw();
```

**Remarks**

Returns true if no conversion will be done.

7.4.1.8.1.5  length

Determines the length between two points.

```cpp
int length(stateT& state, const externT* from,
       const externT* from_end, size_t max) const;
```

**Remarks**

The distance between two points is returned.

7.4.1.8.1.6  max_length

Determines the length necessary for conversion.
int max_length() const throw();

Remarks
The number of elements to convert from externT to internT is returned.

7.4.1.8.1.7 codecvt Virtual Functions

Virtual functions for codecvt implementation.

result do_out(stateT& state, const internT* from,
const internT* from_end,
const internT*& from_next, externT* to,
externT* to_limit, externT*& to_next) const;

Implements out.
The result is returned as a value as in Table 7-3.

result do_in(stateT& state, const externT* from,
const externT* from_end,
const externT*& from_next, internT* to,
internT* to_limit, internT*& to_next) const;

Implements in.
The result is returned as a value as in Table 7-3.

result do_unshift(stateT& state,
externT* to, externT* to_limit, externT*& to_next) const;
Implements unshift.
The result is returned as a value as in Table 7-3.

int do_encoding() const throw();

Implements encoding.

bool do_always_noconv() const throw();

Implements always_noconv.

int do_length(stateT& state, const externT* from, const
externT* from_end, size_t max) const;


Table 7-3. Convert Result Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>Encountered a from_type character it could not convert</td>
</tr>
<tr>
<td>noconv</td>
<td>No conversion was needed</td>
</tr>
<tr>
<td>ok</td>
<td>Completed the conversion</td>
</tr>
<tr>
<td>partial</td>
<td>Not all source characters converted</td>
</tr>
</tbody>
</table>

### 7.4.1.9 Template Class Codecvtbyname

The facet codecvt is responsible for translating internal characters (wchar_t) to/from external char's in a file.

There are several techniques for representing a series of wchar_t's with a series of char's. The codecvtbyname facet can be used to select among several of the encodings. If you construct codecvtbyname with a const char* that refers to a file, then that file is scanned by codecvtbyname's constructor for information to customize the encoding.

```
codecvtbyname<wchar_t, char, std::mbstate_t> cvt("en_US");
```

If the file "en_US" exists, has codecvt data in it, and there are no syntax errors in the data, then cvt will behave as dictated by that data. If the file exists, but does not have codecvt data in it, then the facet will behave as if it were constructed with "C". If the file has codecvt data in it, but there is a syntax error in the data, or if the file does not exist, then a std::runtime_error is thrown.

For codecvtbyname<char, char, mbstate_t>, the codecvt data section begins with:

```
$codecvt_narrow
```

For codecvtbyname<wchar_t, char, mbstate_t>, the codecvt data section begins with:

```
$codecvt_wide
```
Although `$codecvt_narrow` is a valid data section, it really does not do anything. The `codecvtbyname` facet does not add any functionality beyond `codecvt<char, char, mbstate_t>`. This facet is a degenerate case of `noconv` (no conversion). This can be represented in the locale data file as:

```
$codecvt_narrow
   noconv
```

The facet `codecvtbyname<wchar_t, char, mbstate_t>` is much more interesting. After the data section introduction (`$codecvt_wide`), one of these keywords can appear:

- `noconv`
- `UCS-2`
- `JIS`
- `Shift-JIS`
- `EUC`
- `UTF-8`

These keywords will be parsed as strings according to the rules for Strings and Characters in Locale Data Files.

### 7.4.1.10 Codecvtbyname Keywords

These Codecvtbyname keywords will be parsed as strings according to the rules for entering strings in locale data files.

#### 7.4.1.10.1 noconv

This conversion specifies that the base class should handle the conversion. The EWL C++ implementation of `codecvt<wchar_t, char, mbstate_t>` will I/O all bytes of the `wchar_t` in native byte order.

#### 7.4.1.10.2 UCS-2

This encoding input and outputs the two lowest order bytes of the `wchar_t`, high byte first. For a big-endian, 16 bit `wchar_t` platform, this encoding is equivalent to `noconv`. 
### 7.4.1.10.3 JIS

This is an early encoding used by the Japanese to represent a mixture of ASCII and a subset of Kanji.

### 7.4.1.10.4 Shift-JIS

Another early encoding used by the Japanese to represent a mixture of ASCII and a subset of Kanji.

### 7.4.1.10.5 EUC

Extended Unix Code.

### 7.4.1.10.6 UTF-8

A popular Unicode multibyte encoding. For example

```cpp
$codecvt_wide
UTF-8
```

specifies that `codecvt_byname<wchar_t, char, mbstate_t>` will implement the UTF-8 encoding scheme. If this data is in a file called "en_US", then the following program can be used to output a `wchar_t` string in UTF-8 to a file:

**Listing: Example of Writing a wchar_t String in utf-8 to a File:**

```cpp
#include <locale>
#include <fstream>

int main()
{
    std::locale loc("en_US");
    std::wostream out;
    out.imbue(loc);
    out.open("test.dat");
    out << L"This is a test \x00DF";
}
```

The binary contents of the file is (in hex):
Without the UTF-8 encoding, the default encoding will take over (all wchar_t bytes in native byte order):

```cpp
#include <fstream>
int main()
{
    std::wofstream out("test.dat");
    out << L"This is a test \x00DF";
}
```

On a big-endian machine with a 2 byte wchar_t
the resulting file in hex is:

00 54 00 68 00 69 00 73 00 20 00 69 00 73 00 20 00 61 00 20 00 74 00 65 00 73 00 74 00 20 00 DF

00 61 00 20 00 74 00 65 00 73 00 74 00 20 00 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 DF

### 7.4.1.11 Extending codecvt by derivation

The facet codecvt can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons.

Naturally, you can derive from `codecvt` and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the EWL C++ specific classes used to implement `codecvt_byname`. There are five implementation specific facets that you can use in place of `codecvt` or `codecvt_byname` to get the behavior of one of the five encodings:

- `__ucs_2`
- `__jis`
- `__shift_jis`
- `__euc`
- `__utf_8`

These classes are templated simply on the internal character type (and should be instantiated with `wchar_t`). The external character type is implicitly `char`, and the state type is implicitly `mbstate_t`.

Note in An example use of `__utf_8` is: that this locale (and wostream) will have all of the facets of the current global locale except that its `codecvt<wchar_t, char, mbstate_t>` will use the UTF-8 encoding scheme. Thus the binary contents of the file is (in hex):
Listing: An example use of __utf_8 is:

```cpp
#include <locale>
#include <fstream>

int main()
{
    std::locale loc(std::locale(), new std::__utf_8<wchar_t>);
    std::wofstream out;
    out.imbue(loc);
    out.open("test.dat");
    out << L"This is a test \x00DF";
}
```

Result

```
54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 C3 9F
```

7.4.2 The Numeric Category

A class for numeric formatting and manipulation for locales.

7.4.2.1 Template Class Num_get

A class for formatted numeric input.

7.4.2.2 Num_get Members

The class num_get includes specific functions for parsing and formatting of numbers.

7.4.2.2.1 get

The function `get` is overloaded for un-formatted input.

```
iter_type get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err,long& val) const;
iter_type get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err,double& val) const;
```
Remarks

returns and iterator type.

7.4.2.2.2 Num_get Virtual Functions

Implements the relative versions of the get function

Remarks

Implements the relative versions of get.

TemplateClassNum_put

A class for formatted numeric output.
7.4.2.3 Num_put Members

The class `num_put` includes specific functions for parsing and formatting of numbers.

7.4.2.3.1 put

The function `put` is overloaded for un-formatted output.

```cpp
iter_type put(iter_type out, ios_base& str,
        char_type fill, bool val) const;
iter_type put(iter_type out, ios_base& str,
        char_type fill, long val) const;
iter_type put(iter_type out, ios_base& str,
        char_type fill, unsigned long val) const;
iter_type put(iter_type out, ios_base& str,
        char_type fill, double val) const;
iter_type put(iter_type out, ios_base& str,
        char_type fill, long double val) const;
iter_type put(iter_type out, ios_base& str,
        char_type fill const void* val) const;
```

7.4.2.3.2 Num_put Virtual Functions

Implementation functions for `put`.

```cpp
iter_type do_put(iter_type out, ios_base& str,
        char_type fill, bool val) const;
iter_type do_put(iter_type out, ios_base& str,
        char_type fill, long val) const;
iter_type do_put(iter_type out, ios_base& str,
        char_type fill, unsigned long val) const;
iter_type do_put(iter_type out, ios_base& str,
        char_type fill, double val) const;
iter_type do_put(iter_type out, ios_base& str,
        char_type fill, long double val) const;
iter_type do_put(iter_type out, ios_base& str,
        char_type fill, const void* val) const;
```

7.4.2.4 The Numeric Punctuation Facet

A facet for numeric punctuation in formatting and parsing.

TemplateClassNumpunct
A class for numeric punctuation conversion.

### 7.4.2.4.1 Numpunct Members

The template class numpunct provides various functions for punctuation localizations.

#### 7.4.2.4.1.1 decimal_point

Determines the character used for a decimal point.

```cpp
char_type decimal_point() const;
```

**Remarks**

Returns the character used for a decimal point.

#### 7.4.2.4.1.2 thousands_sep

Determines the character used for a thousand separator.

```cpp
char_type thousands_sep() const;
```

**Remarks**

Returns the character used for the thousand separator.

#### 7.4.2.4.1.3 grouping

Describes the thousand separators.

```cpp
string grouping() const;
```

**Remarks**

Returns a string describing the thousand separators.

#### 7.4.2.4.1.4 truename

Determines the localization for "true".
string_type truename() const;

Remarks
Returns a string describing the localization of the word "true".

7.4.2.4.1.5 falsename

Determines the localization for "false".

string_type falsename() const;

Remarks
Returns a string describing the localization of the word "false".

7.4.2.4.1.6 numpunct virtual functions

Implementation of the public functions.

char_type do_decimal_point() const;

Implements decimal_point.

string_type do_thousands_sep() const;

Implements thousands_sep.

string do_grouping() const;

Implements grouping.

string_type do_truename() const;

Implements truename.

string_type do_falsename() const;

Implements falsename.
7.4.2.4.1.7 Template Class Numpunct_byname

The facet numpunct specifies the punctuation used for parsing and formatting numeric quantities. You can specify the decimal point character, thousands separator, the grouping, and the spelling of true and false. If you construct numpunct_byname with a const char* that refers to a file, then that file is scanned by numpunct_byname’s constructor for information to customize the encoding.

    numpunct_byname<char> np("en_US");

If the file "en_US" exists, has numpunct data in it, and there are no syntax errors in the data, then np will behave as dictated by that data. If the file exists, but does not have numpunct data in it, then the facet will behave as if it were constructed with "C". If the file has numpunct data in it, but there is a syntax error in the data, or if the file does not exist, then a std::runtime_error is thrown.

For numpunct_byname<char>, the numpunct data section begins with:

    $numeric_narrow

For numpunct_byname<wchar_t>, the numpunct data section begins with:

    $numeric_wide

The syntax for both the narrow and wide data sections is the same. There are keywords that allow you to specify the different parts of the numpunct data:

- decimal_point
- thousands_sep
- grouping
- false_name and true_name

You enter data with one of these keywords, followed by an equal sign '=', and then the data. You can specify any or all of the keywords. Data not specified will default to that of the "C" locale. The first two keywords (decimal_point and thousands_sep) have character data associated with them. See the rules for Character Syntax for details. The last three keywords have string data associated with them. See the rules for String Syntax.

Listing: Example usage of numpunct_byname

    $numeric_narrow
decimal_point = ','
thousands_sep = '.'

grouping = 3|2
false_name = nope
true_name = sure

Here is an example program using the above data for narrow streams:

```cpp
#include <sstream>
#include <locale>
#include <iostream>
int main()
{
    std::locale loc("my_loc");
    std::cout.imbue(loc);
    std::istringstream in("1.23.456 nope 1.23.456,789");
    in.imbue(loc);
    in >> std::boolalpha;
    long i;
    bool b;
    double d;
    in >> i >> b >> d;
    std::cout << i << '
'      << std::boolalpha << !b << '
'      << std::fixed << d;
}
```

The output is:

```
1.23.456
sure
1.23.456,789000
```

### 7.4.2.4.1.7.1 \textit{decimal_point}

The decimal point data is a single character, as in:

```
decimal_point = '.'
```

### 7.4.2.4.1.7.2 \textit{thousands_sep}

The character to be used for the thousands separator is specified with \textit{thousands_sep}, as in:

```
thousands_sep = ','
```

### 7.4.2.4.1.7.3 \textit{grouping}

...
The grouping string specifies the number of digits to group, going from right to left. For example, the grouping: 321 means that the number 12345789 would be printed as in:

1,2,3,4,56,789

The above grouping string can be specified as:

grouping = 321

A grouping string of "0" or "" means: do not group.

7.4.2.4.1.7.4  false_name and true_name

The names of false and true can be specified with false_name and true_name. For example:

false_name = "no way"

true_name = sure

7.4.2.4.1.8  Numeric_wide

For $\text{numeric\_wide}$, wide characters can be represented with the hex or universal format (e.g. "\u64D0").

7.4.2.5  Extending numpunct by derivation

It is easy enough to derive from numpunct and override the virtual functions in a portable manner. But numpunct also has a non-standard protected interface that you can take advantage of if you wish.

There are five protected data members:

```cpp
char_type __decimal_point;
char_type __thousands_sep;
string __grouping;
```
A derived class could set these data members in its constructor to whatever is appropriate, and thus not need to override the virtual methods.

**Listing: Example of numpunct<char>**

```cpp
struct mypunct: public std::numpunct<char>
{
    mypunct();
};

mypunct::mypunct()
{
    __decimal_point_ = ',';
    __thousands_sep_ = '.';
    __grouping_ = "\3\2";
    __falsename_ = "nope";
    __truename_ = "sure";
}
```

```cpp
int main()
{
    std::locale loc(std::locale(), new mypunct);
    std::cout.imbue(loc);
    // ...
}
```

### 7.4.3 The Collate Category

The Template class collate used for the comparison and manipulation of strings.

#### 7.4.3.1 Collate Members

Member functions used for comparison and hashing of strings.

#### 7.4.3.1.1 compare
Lexicographical comparison of strings.

```cpp
int compare(const charT* low1, const charT* high1,
            const charT* low2, const charT* high2) const;
```

**Remarks**

A value of 1 is returned if the first is lexicographically greater than the second. A value of negative 1 is returned if the second is greater than the first. A value of zero is returned if the strings are the same.

### 7.4.3.1.2 transform

Provides a string object to be compared to other transformed strings.

```cpp
string_type transform
   (const charT* low, const charT* high) const;
```

**Remarks**

The `transform` member function is used for comparison of a series of strings. Returns a string for comparison.

### 7.4.3.1.3 hash

Determines the `hash` value for the string.

```cpp
long hash(const charT* low, const charT* high) const;
```

**Remarks**

Returns the hash value of the string.

### 7.4.3.1.4 collate Virtual Functions

Localized implementation functions for public collate member functions.

```cpp
int do_compare
   (const charT* low1, const charT* high1,
    const charT* low2, const charT* high2) const;
```
Implements `compare`.

```cpp
    string_type do_transform(const charT* low, const charT* high) const;
```

Implements `transform`:

```cpp
    long do_hash(const charT* low, const charT* high) const;
```

Implements `hash`.

### 7.4.3.2 Template Class Collate_byname

The facet `collate` is responsible for specifying the sorting rules used for sorting strings.

The base class `collate` does a simple lexical comparison on the binary values in the string. `collate_byname` can perform much more complex comparisons that are based on the Unicode sorting algorithm. If you construct `collate_byname` with a `const char*` that refers to a file, then that file is scanned by `collate_byname`'s constructor for information to customize the collation rules.

```cpp
    collate_byname<char> col("en_US");
```

If the file "en_US" exists, has collate data in it, and there are no syntax errors in the data, then `col` will behave as dictated by that data. If the file exists, but does not have collate data in it, then the facet will behave as if it were constructed with "C". If the file has collate data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

#### 7.4.3.2.1 Collate Data Section

For `collate_byname<char>`, the collate data section begins with:

```
$collate_narrow
```

For `collate_byname<wchar_t>`, the collate data section begins with:

```
$collate_wide
```
The syntax for both the narrow and wide data sections is the same. The data consists of a single string that has a syntax very similar to Java's RuleBasedCollator class. This syntax is designed to provide a level three sorting key consistent with the sorting algorithm specified by the Unicode collation algorithm.

7.4.3.2.2 Rule Format

The collation string rule is composed of a list of collation rules, where each rule is of three forms:

```
< modifier >
< relation > < text-argument >
< reset >   < text-argument >
```

7.4.3.2.3 Text-Argument:

A text-argument is any sequence of characters, excluding special characters (that is, common whitespace characters and rule syntax characters. If those characters are desired, you can put them in single quotes (e.g. ampersand => '&').

7.4.3.2.4 Modifier:

There is a single modifier which is used to specify that all accents (secondary differences) are backwards.

'@': Indicates that accents are sorted backwards, as in French.

7.4.3.2.5 Relation:

The relations are the following:

- '<': Greater, as a letter difference (primary)
- ':' Greater, as an accent difference (secondary)
- ',' Greater, as a case difference (tertiary)
- '=': Equal

7.4.3.2.6 Reset:
There is a single reset which is used primarily for expansions, but which can also be used to add a modification at the end of a set of rules.

- '&': Indicates that the next rule follows the position to where the reset text-argument would be sorted.

### 7.4.3.2.7 Relationals

The relationals allow you to specify the relative ordering of characters. For example, the following string expresses that 'a' is less than 'b' which is less than 'c':

```
"< a < b < c"
```

For the time being, just accept that a string should start with '<'. That rule will be both relaxed and explained later.

Many languages (including English) consider 'a' < 'A', but only as a tertiary difference. And such minor differences are not considered significant unless more important differences are found to be equal. For example consider the strings:

- aa
- Aa
- ab

Since 'a' < 'A', then "aa" < "Aa". But "Aa" < "ab" because the difference between the second characters 'a' and 'b' is more important the difference between the first characters 'A' and 'a'. This type of relationship can be expressed in the collation rule with:

```
"< a, A < b, B < c, C"
```

This says that 'a' is less 'A' by a tertiary difference, and then 'b' and 'B' are greater than 'a' and 'A' by a primary difference (similarly for 'c' and 'C').

Accents are usually considered secondary differences. For example, lower case e with an acute accent might be considered to be greater than lower case e, but only by a secondary difference. This can be represented with a semicolon like:

```
"... < e, E ; é, É < ...
```

Note that characters can be entered in hexadecimal or universal format. They can also be quoted with single quotes (for example 'a'). If it is ambiguous whether a character is a command or a text argument, adding quotes specifies that it is a text argument.
Characters not present in a rule are implicitly ordered after all characters that do appear in a rule.

### 7.4.3.2.8 French collation

Normally primary, secondary and tertiary differences are considered left to right. But in French, secondary differences are considered right to left. This can be specified in the rule string by starting it with '@':

```
"@ ... < e, E ; é, ŋ < ..."
```

### 7.4.3.2.9 Contraction

Some languages sort groups of letters as a single character. Consider the two strings: "acha" and "acia". In English they are sorted as just shown. But Spanish requires "ch" to be considered a single character that is sorted after 'c' and before 'd'. Thus the order in Spanish is reversed relative to English (that is "acia" < "acha"). This can be specified like:

```
"... < c < ch < d ..."
```

Taking case into account, you can expand this idea to:

```
"... < c, C < ch, cH, Ch, CH < d, D ..."
```

### 7.4.3.2.10 Expansion

Some languages expand a single character into multiple characters for sorting. For example in English the ligature 'æ' might be sorted as 'a' followed by 'e'. To represent this in a rule, the reset character (&) is used. The idea is to reset the current sorting key to an already entered value, and create multiple entries for the ligature. For example:

```
"... < a < b < c < d < e ... < z & a = æ & e = æ ...
```

This rule resets the sort key to that of 'a', and then enters 'æ'. Then resets the sort key to that of 'e' and enters 'æ' again. This rule says that 'æ' is exactly equivalent to 'a' followed by 'e'. Alternatively ',' could have been used instead of '='. This would have made "æ" less than "æ" but only by a secondary difference.
### Ignorable Characters

Characters in the rule before the first `<` are ignorable. They are not considered during the primary sorting. Accents and punctuation are often marked as ignorable, but given a non-ignorable secondary or tertiary weight. For example, the default Java rule starts out with:

```
"="\u200B="\u200C="\u200D="\u200E="\u200F ...
";"\u0020";"\u00A0"...
```

This completely ignores the first five characters (formatting control), and ignores except for secondary differences the next two characters (spacing characters).

This is why all example rules up till now started with `<` (so that none of the characters would be ignorable).

In the notice how the space character was entered using quotes to disambiguate it from insignificant white space. Example of locale sorting notice how the space character was entered using quotes to disambiguate it from insignificant white space. Example of locale sorting

Assume the file "my_loc" has the following data in it:

```
$collate_narrow

"; = ' 
< a, A < b, B < c, C
< ch, cH, Ch, CH
< d, D < e, E < f, F
< g, G < h, H < i, I
< j, J < k, K < l, L
< ll, lL, Ll, LL
< m, M < n, N < o, O
< p, P < q, Q < r, R
< s, S < t, T < u, U
< v, V < w, W < x, X
< y, Y < z, Z
```

The program below creates a vector of strings and sorts them both by "binary order" (just using string's operator `<`), and by the custom rule above using a locale as the sorting key.
#include <locale>
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>

int main()
{
    std::vector<std::string> v;
    v.push_back("aaaaaaB");
    v.push_back("aaaaaaA");
    v.push_back("AaaaaaB");
    v.push_back("AaaaaaA");
    v.push_back("blackbird");
    v.push_back("black-bird");
    v.push_back("black bird");
    v.push_back("blackbirds");
    v.push_back("acia");
    v.push_back("acha");
    std::ostream_iterator<std::string> out(std::cout, "\n");
    std::cout << "Binary order:\n"
    std::sort(v.begin(), v.end());
    std::copy(v.begin(), v.end(), out);
    std::cout << '
';
    std::locale loc("my_loc");
    std::sort(v.begin(), v.end(), loc);
    std::cout << "Customized order:\n"
    std::copy(v.begin(), v.end(), out);
    std::cout << '
';
}

The output is:

Binary order:

AaaaaaA
AaaaaaB
aaaaaaA
aaaaaaB
acha
acia
black bird
black-bird
blackbird
blackbirds

Customized order:

aaaaaaA
AaaaaaA
aaaaaaB
AaaaaaB
acia
acha
blackbird
black-bird
black bird
blackbirds
7.4.3.3 Extending collate by derivation

The behavior of collate can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons.

Naturally, you can derive from collate and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the EWL C++ specific protected interface of collate_byname if you wish (to make your job easier if portability is not a concern).

The class collate_byname has one protected data member:

```cpp
__collation_rule<charT> rule_;
```

Listing: The class std::__collation_rule interface:

```cpp
template <class charT>
class __collation_rule
{
  struct value
  {
    charT primary;
    charT secondary;
    charT tertiary;
  }
  public:
    struct entry : value
    {
      unsigned char length;
    };
    __collation_rule();
    explicit __collation_rule(const basic_string<charT>& rule);
    void set_rule(const basic_string<charT>& rule);
    entry operator()(const charT* low,
                     const charT* high, int& state) const;
    bool is_french() const;
    bool empty() const;
}```
Most of this interface is to support `collatebyname`. If you simply derive from `collatebyname`, set the rule with a string, and let `collatebyname` do all the work, then there is really very little you have to know about `__collation_rule`.

A `__collation_rule` can be empty (contain no rule). In that case `collatebyname` will use `collate`'s sorting rule. This is also the case if `collatebyname` is constructed with "C". And once constructed, `__collation_rule`'s rule can be set or changed with `set_rule`. That is all you need to know to take advantage of all this horsepower!

**Listing: Example of a __collation_rule:**

```cpp
#include <iostream>
#include <locale>
#include <string>

struct my_collate
    : public std::collatebyname<char>
{
    my_collate();

    my_collate::my_collate()
        : std::collatebyname<char>("C")
    {
        rule_.set_rule("< a = A < b = B < c = C
            < d = D < e = E < f = F"
            "< g = G < h = H < i = I"
            "< j = J < k = K < l = L"
            "< m = M < n = N < o = O"
            "< p = P < q = Q < r = R"
            "< s = S < t = T < u = U"
            "< v = V < w = W < x = X"
            "< y = Y < z = Z");
    }

    int main()
    {
        std::locale loc(std::locale(), new my_collate);
```
std::string s1("Arnold");
std::string s2("arnold");
if (loc(s1, s2))
    std::cout << s1 << " < " << s2 << '
';
else if (loc(s2, s1))
    std::cout << s1 << " > " << s2 << '
';
else
    std::cout << s1 << " == " << s2 << '
';
}

The custom facet my_collate derives from std::collatebyname<char> and sets the rule in its constructor. That's all it has to do. For this example, a case-insensitive rule has been constructed. The output of this program is:

Arnold == arnold

Alternatively, you could use my_collate directly (this is exactly what EWL C++'s locale does):

Listing: Example of custom facet my_collate:

int main()
{
    my_collate col;
    std::string s1("Arnold");
    std::string s2("arnold");
    switch (col.compare(s1.data(), s1.data()+s1.size(),
                        s2.data(), s2.data()+s2.size())
    {
    case -1:
        std::cout << s1 << " < " << s2 << '
';
        break;
    case  0:
        std::cout << s1 << " == " << s2 << '
';
        break;
    case  1:
        std::cout << s1 << " > " << s2 << '
';
    }
The output of this program is also:

Arnold == arnold

7.4.4 The Time Category

The facets `time_get` and `time_put` are conceptually simple: they are used to parse and format dates and times in a culturally sensitive manner. But as is not uncommon, there can be a lot of details. And for the most part, the standard is quiet about the details, leaving much of the behavior of these facets in the "implementation defined" category. Therefore this document not only discusses how to extend and customize the time facets, but it also explains much of the default behavior as well.

7.4.4.1 Time_get Members

The facet `time_get` has 6 member functions.

- `date_order`
- `get_time`
- `get_date`
- `get_weekday`
- `get_monthname`
- `get_year`

```cpp
date_order date_order() const;
```

Determines how the date, month and year are ordered.

Returns an enumeration representing the date, month, year order. Returns zero if it is unordered.

```cpp
iter_type get_time
(iter_type s, iter_type end, ios_base& str,
 ios_base::iostate& err, tm* t) const;
```
Determines the localized time.

Returns an iterator immediately beyond the last character recognized as a valid time.

iter_type get_date
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;

Determines the localized date.

Returns an iterator immediately beyond the last character recognized as a valid date.

iter_type get_weekday
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;

Determines the localized weekday.

Returns an iterator immediately beyond the last character recognized as a valid weekday.

iter_type get_monthname
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;

Determines the localized month name.

Returns an iterator immediately beyond the last character recognized as a valid month name.

iter_type get_year(iter_type s, iter_type end,
ios_base& str, ios_base::iostate& err,
tm* t) const;

Determines the localized year.

Returns an iterator immediately beyond the last character recognized as a valid year.

### 7.4.4.2 Time_get Virtual Functions

The facet time_get has 6 protected virtual members.

- do_date_order
- do_get_time
- do_get_date
- do_get_weekday
• \texttt{do\_get\_monthname}

• \texttt{do\_get\_year}

\begin{verbatim}
dateorder
    do\_date\_order() const;
\end{verbatim}

The method \texttt{do\_date\_order} returns \texttt{no\_order}. This result can be changed via derivation.

\begin{verbatim}
iter\_type do\_get\_time(iter\_type s, iter\_type end,
    ios\_base\& str, ios\_base::iostate\& err,
    tm\* t) const;
\end{verbatim}

The method \texttt{do\_get\_time} parses time with the format:

\begin{verbatim}
"%H:%M:%S"
\end{verbatim}

\begin{verbatim}
iter\_type do\_get\_date
    (iter\_type s, iter\_type end, ios\_base\& str,
    ios\_base::iostate\& err, tm\* t) const;
\end{verbatim}

The method \texttt{do\_get\_date} parses a date with the format:

\begin{verbatim}
"%A %B %d %T %Y"
\end{verbatim}

This format string can be changed via the named locale facility, or by derivation.

\begin{verbatim}
iter\_type do\_get\_weekday
    (iter\_type s, iter\_type end, ios\_base\& str,
    ios\_base::iostate\& err, tm\* t) const;
\end{verbatim}

The method \texttt{do\_get\_weekday} parses with the format:

\begin{verbatim}
"%A"
\end{verbatim}

Although the format string can only be changed by derivation, the names of the weekdays themselves can be changed via the named locale facility or by derivation.

\begin{verbatim}
iter\_type do\_get\_monthname
    (iter\_type s, iter\_type end, ios\_base\& str,
    ios\_base::iostate\& err, tm\* t) const;
\end{verbatim}

The method \texttt{do\_get\_monthname} parses with the format:

\begin{verbatim}
"%B"
\end{verbatim}
Although the format string can only be changed by derivation, the names of the months themselves can be changed via the named locale facility or by derivation.

```cpp
iter_type do_get_year
  (iter_type s, iter_type end, ios_base& str,
   ios_base::iostate& err, tm* t) const;
```

The method `do_get_year` parses a year with the format:

```
%Y
```

This behavior can only be changed by derivation.

The details of what these formats mean can be found in the Format/Parsing Table.

In addition to the above mentioned protected methods, **EWL C++** provides a non-standard, non-virtual protected method:

```cpp
iter_type __do_parse(iter_type in, iter_type end,
                     ios_base& str, ios_base::iostate& err,
                     const basic_string<charT>& pattern, tm* t) const;
```

This method takes the parameters typical of the standard methods, but adds the pattern parameter of type `basic_string`. The pattern is a general string governed by the rules outlined in the section **Format Parsing**. Derived classes can make use of this method to parse patterns not offered by `time_get`.

**Listing: Derived classes example:**

```cpp
template <class charT, class InputIterator>
typedef my_time_get<charT, InputIterator>::iter_type
  my_time_get<charT, InputIterator>::do_get_date_time(
    iter_type in, iter_type end, std::ios_base& str,
    std::ios_base::iostate& err, std::tm* t) const
{
  const std::ctype<charT>& ct = std::use_facet<std::ctype<charT>>
    (str.getloc());
  return __do_parse(in, end, str, err, ct.widen("%c"), t);
}
```

**7.4.4.3 Format Parsing**

These commands follow largely from the C90 and C99 standards.
However a major difference here is that most of the commands have meaning for parsing as well as formatting, whereas the C standard only uses these commands for formatting. The pattern string consists of zero or more conversion specifiers and ordinary characters (char or wchar_t). A conversion specifier consists of a % character, possibly followed by an E or O modifier character (described below), followed by a character that determines the behavior of the conversion specifier. Ordinary characters (non-conversion specifiers) must appear in the source string during parsing in the appropriate place or failbit gets set. On formatting, ordinary characters are sent to the output stream unmodified.

The E modifier can appear on any conversion specifier. But it is ignored for both parsing and formatting.

The O modifier can appear on any conversion specifier. It is ignored for parsing, but effects the following conversion specifiers on output by not inserting leading zeroes: %c, %d, %D, %f, %g, %H, %I, %j, %m, %u, %v, %w, %y

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Parse</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Reads one of the locale's weekday names. The name can either be the full name, or the abbreviated name. Case is significant. On successful parsing of one of the weekday names, sets tm_wday, otherwise sets failbit. For parsing, this format is identical to %A.</td>
<td>Outputs the locale's abbreviated weekday name as specified by tm_wday. The &quot;C&quot; locale's abbreviated weekday names are: Sun, Mon, Tue, Wed, Thu, Fri, Sat.</td>
</tr>
<tr>
<td>%A</td>
<td>For parsing, this format is identical to %a.</td>
<td>Outputs the locale's full weekday name as specified by tm_wday. The &quot;C&quot; locale's full weekday names are: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.</td>
</tr>
<tr>
<td>%b</td>
<td>Reads one of the locale's month names. The name can either be the full name, or the abbreviated name. Case is significant. On successful parsing of one of the month names, sets tm_mon, otherwise sets failbit. For parsing, this format is identical to %B.</td>
<td>Outputs the locale's abbreviated month name as specified by tm_mon. The &quot;C&quot; locale's abbreviated month names are: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec.</td>
</tr>
<tr>
<td>%B</td>
<td>For parsing, this format is identical to %b.</td>
<td>Outputs the locale's full month name as specified by tm_mon. The &quot;C&quot; locale's full month names are: January, February, March, April, May, June, July, August, September, October, November, December.</td>
</tr>
<tr>
<td>%c</td>
<td>Reads the date-and-time as specified by the current locale. The &quot;C&quot; locale specification is &quot;%A %B %d %T %Y&quot;. On successful parsing this sets tm_wday, tm_mon, tm_mday, tm_sec, tm_min, tm_hour, tm_year.</td>
<td>Outputs the locale's date-and-time. The &quot;C&quot; locale's date-and-time format is &quot;%A %B %d %T %Y&quot;. This information is specified by tm_wday, tm_mon, tm_mday, tm_sec, tm_min, tm_hour and tm_year.</td>
</tr>
</tbody>
</table>

Table 7-4. Format/Parsing Table

Table continues on the next page...
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Parse</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>tm_hour and tm_year. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set.</td>
<td>Outputs the current year divided by 100. Single digit results will be pre-appended with '0' unless the O modifier is used.</td>
<td></td>
</tr>
<tr>
<td>%C</td>
<td>This is not a valid parse format. If %C is used in a parse pattern, a runtime_error is thrown.</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Reads the day of the month. The result must be in the range [1, 31] else failbit will be set. Upon successful parsing tm_mday is set. For parsing, this format is identical to %e.</td>
<td>Outputs the day of the month as specified by tm_mday. Single digit results will be pre-appended with '0' unless the O modifier is used.</td>
</tr>
<tr>
<td>%D</td>
<td>Is equivalent to &quot;%m/%d/%y&quot;. If the O modifier is used, is equivalent to &quot;%Om/%Od/%y&quot;.</td>
<td></td>
</tr>
<tr>
<td>%e</td>
<td>Reads the day of the month. The result must be in the range [1, 31] else failbit will be set. Upon successful parsing tm_mday is set. For parsing, this format is identical to %d.</td>
<td>Outputs the day of the month as specified by tm_mday. Single digit results will be pre-appended with a space.</td>
</tr>
<tr>
<td>%F</td>
<td>Is equivalent to &quot;%Y-%m-%d&quot; (the ISO 8601 date format). Is equivalent to &quot;%Y-%m-%d&quot;. If the O modifier is used, is equivalent to &quot;%Y-%Om-%Od&quot;.</td>
<td></td>
</tr>
<tr>
<td>%g</td>
<td>This is not a valid parse format. If %g is used in a parse pattern, a runtime_error is thrown.</td>
<td>Outputs the last 2 digits of the ISO 8601 week-based year. Single digit results will be pre-appended with '0' unless the O modifier is used. Specified by tm_year, tm_wday and tm_yday.</td>
</tr>
<tr>
<td>%G</td>
<td>This is not a valid parse format. If %G is used in a parse pattern, a runtime_error is thrown.</td>
<td>Outputs the ISO 8601 week-based year. Specified by tm_year, tm_wday and tm_yday.</td>
</tr>
<tr>
<td>%h</td>
<td>Is equivalent to %b.</td>
<td>Is equivalent to %b.</td>
</tr>
<tr>
<td>%H</td>
<td>Reads the hour (24-hour clock) as a decimal number. The result must be in the range [0, 23] else failbit will be set. Upon successful parsing tm_hour is set.</td>
<td>Outputs the hour (24-hour clock) as specified by tm_hour. Single digit results will be pre-appended with '0' unless the O modifier is used.</td>
</tr>
<tr>
<td>%I</td>
<td>Reads the hour (12-hour clock) as a decimal number. The result must be in the range [1, 12] else failbit will be set. Upon successful parsing tm_hour is set. This format is usually used with %p to specify am/pm. If a %p is not parsed with the %I, am is assumed.</td>
<td>Outputs the hour (12-hour clock) as specified by tm_hour. Single digit results will be pre-appended with '0' unless the O modifier is used.</td>
</tr>
<tr>
<td>%j</td>
<td>This is not a valid parse format. If %j is used in a parse pattern, a runtime_error is thrown.</td>
<td>Outputs the day of the year as specified by tm_yday in the range [001, 366]. If the O modifier is used, leading zeroes are suppressed.</td>
</tr>
<tr>
<td>%m</td>
<td>Reads the month as a decimal number. The result must be in the range [1, 12] else failbit will be set. Upon successful parsing tm_mon is set.</td>
<td>Outputs the month as specified by tm_mon as a decimal number in the range [1, 12]. Single digit results will be pre-appended with '0' unless the O modifier is used.</td>
</tr>
</tbody>
</table>
Table 7-4. Format/Parsing Table (continued)

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Parse</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>%M</td>
<td>Reads the minute as a decimal number. The result must be in the range [0, 59] else failbit will be set. Upon successful parsing tm_min is set.</td>
<td>Outputs the minute as specified by tm_min as a decimal number in the range [0, 59]. Single digit results will be pre-appended with '0' unless the O modifier is used.</td>
</tr>
<tr>
<td>%n</td>
<td>Is equivalent to '\n'. A newline must appear in the source string at this position else failbit will be set.</td>
<td>Is equivalent to '\n'. A newline is output.</td>
</tr>
<tr>
<td>%p</td>
<td>Reads the locale's designation for am or pm. If neither of these strings are parsed then failbit will be set. A successful read will modify tm_hour, but only if %I is successfully parsed in the same parse pattern.</td>
<td>Outputs the locale's designation for am or pm, depending upon the value of tm_hour. The &quot;C&quot; locale's designations are am and pm.</td>
</tr>
<tr>
<td>%r</td>
<td>Reads the 12-hour time as specified by the current locale. The &quot;C&quot; locale specification is &quot;%I:%M:%S %p&quot;. On successful parsing this sets tm_hour, tm_min, and tm_sec. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set.</td>
<td>Outputs the locale's 12-hour time. The &quot;C&quot; locale's date-and-time format is &quot;%I:%M:%S %p&quot;. This information is specified by tm_hour, tm_min, and tm_sec.</td>
</tr>
<tr>
<td>%R</td>
<td>Is equivalent to &quot;%H:%M&quot;.</td>
<td>Is equivalent to &quot;%H:%M&quot;. If the O modifier is used, is equivalent to &quot;%OH: %M&quot;.</td>
</tr>
<tr>
<td>%S</td>
<td>: Reads the second as a decimal number. The result must be in the range [0, 60] else failbit will be set. Upon successful parsing tm_sec is set.</td>
<td>Outputs the second as specified by tm_sec as a decimal number in the range [0, 60]. Single digit results will be pre-appended with '0' unless the O modifier is used.</td>
</tr>
<tr>
<td>%t</td>
<td>Is equivalent to '\t'. A tab must appear in the source string at this position else failbit will be set.</td>
<td>Is equivalent to '\t'. A tab is output.</td>
</tr>
<tr>
<td>%T</td>
<td>Is equivalent to &quot;%H:%M:%S&quot;.</td>
<td>Is equivalent to &quot;%H:%M:%S&quot;. If the O modifier is used, is equivalent to &quot;%OH: %M:%S&quot;.</td>
</tr>
<tr>
<td>%u</td>
<td>Reads the ISO 8601 weekday as a decimal number [1, 7], where Monday is 1. If the result is outside the range [1, 7] failbit will be set. Upon successful parsing tm_wday is set.</td>
<td>Outputs tm_wday as the ISO 8601 weekday in the range [1, 7] where Monday is 1.</td>
</tr>
<tr>
<td>%U</td>
<td>This is not a valid parse format. If %U is used in a parse pattern, a runtime_error is thrown.</td>
<td>Outputs the week number of the year (the first Sunday as the first day of week 1) as a decimal number in the range [00, 53] using tm_year, tm_wday and tm_yday. If the O modifier is used, any leading zero is suppressed.</td>
</tr>
<tr>
<td>%V</td>
<td>This is not a valid parse format. If %V is used in a parse pattern, a runtime_error is thrown.</td>
<td>Outputs the ISO 8601 week-based year week number in the range [01, 53]. Specified by tm_year, tm_wday and tm_yday. If the O modifier is used, any leading zero is suppressed.</td>
</tr>
</tbody>
</table>
Table 7-4. Format/Parsing Table (continued)

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Parse</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>%w</td>
<td>Reads the weekday as a decimal number [0, 6], where Sunday is 0. If the result is outside the range [0, 6] failbit will be set. Upon successful parsing tm_wday is set.</td>
<td>Outputs tm_wday as the weekday in the range [0, 6] where Sunday is 0.</td>
</tr>
<tr>
<td>%W</td>
<td>This is not a valid parse format. If %W is used in a parse pattern, a runtime_error is thrown.</td>
<td>Outputs the week number in the range [00, 53]. Specified by tm_year, tm_wday and tm_yday. The first Monday as the first day of week 1. If the O modifier is used, any leading zero is suppressed.</td>
</tr>
<tr>
<td>%x</td>
<td>Reads the date as specified by the current locale. The &quot;C&quot; locale specification is &quot;%A %B %d %Y&quot;. On successful parsing this sets tm_wday, tm_mon, tm_mday, and tm_year. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set.</td>
<td>Outputs the locale’s date. The &quot;C&quot; locale's date format is &quot;%A %B %d %Y&quot;. This information is specified by tm_wday, tm_mon, tm_mday, and tm_year.</td>
</tr>
<tr>
<td>%X</td>
<td>Reads the time as specified by the current locale. The &quot;C&quot; locale specification is &quot;%H:%M:%S&quot;. On successful parsing this sets tm_hour, tm_min, and tm_sec. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set.</td>
<td>Outputs the locale’s time. The &quot;C&quot; locale’s time format is &quot;%H:%M:%S&quot;. This information is specified by tm_hour, tm_min, and tm_sec.</td>
</tr>
<tr>
<td>%y</td>
<td>Reads the year as a 2 digit number. The century is specified by the locale. The &quot;C&quot; locale specification is 20 (the 21st century). On successful parsing this sets tm_year. If the year is not successfully parsed, then tm_year is not set and failbit is set.</td>
<td>Outputs the last two digits of tm_year. Single digit results will be pre-appended with '0' unless the O modifier is used.</td>
</tr>
<tr>
<td>%Y</td>
<td>Reads the year. On successful parsing this sets tm_year. If the year is not successfully parsed, then tm_year is not set and failbit is set.</td>
<td>Outputs the year as specified by tm_year. (e.g. 2001)</td>
</tr>
<tr>
<td>%z</td>
<td>Reads the offset from UTC in the ISO 8601 format &quot;-0430&quot; (meaning 4 hours 30 minutes behind UTC, west of Greenwich). Two strings are accepted according to the current locale, one indicating Daylight Savings Time is not in effect, the other indicating it is in effect. Depending upon which string is read, tm_isdst will be set to 0 or 1. If the locale's designations for these strings are zero length, then no parsing is done and tm_isdst is set to -1. If the locale has non-empty strings for the UTC offset and neither is successfully parsed, failbit is set.</td>
<td>Outputs the UTC offset according to the current locale and the setting of tm_isdst (if non-negative). The &quot;C&quot; locale’s designation for these strings is &quot;&quot; (an empty string).</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 7-4. Format/Parsing Table (continued)

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Parse</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Z</td>
<td>: Reads the time zone name. Two strings are accepted according to the current locale, one indicating Daylight Savings Time is not in effect, the other indicating it is in effect. Depending upon which string is read, <code>tm_isdst</code> will be set to 0 or 1. If the locale's designations for these strings are zero length, then no parsing is done and <code>tm_isdst</code> is set to -1. If the locale has non-empty strings for the time zone names and neither is successfully parsed, failbit is set.</td>
<td>Outputs the time zone according to the current locale and the setting of <code>tm_isdst</code> (if non-negative). The &quot;C&quot; locale's designation for these strings is &quot;&quot; (an empty string).</td>
</tr>
<tr>
<td>%%</td>
<td>A % must appear in the source string at this position else failbit will be set</td>
<td>A % is output.</td>
</tr>
<tr>
<td>% followed by a space</td>
<td>One or more white space characters are parsed in this position. White space is determined by the locale's ctype facet. If at least one white space character does not exist in this position, then failbit is set.</td>
<td>A space (&quot; &quot;) for output.</td>
</tr>
</tbody>
</table>

### 7.4.4.4 ISO 8601 week-based year

The `%g`, `%G`, and `%V` give values according to the ISO 8601 week-based year.

In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, `%G` is replaced by 1998 and `%V` is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, `%G` is replaced by 1998 and `%V` is replaced by 1.

### 7.4.4.5 Template Class `Time_get_byname`

A class used for locale time manipulations.

**Listing: Template class `time_get_byname`**

```cpp
namespace std {
    template <class charT,
```
class InputIterator = istreambuf_iterator<charT> >
class time_get_byname
    : public time_get<charT, InputIterator>
{
public:
    typedef time_base::dateorder dateorder;
    typedef InputIterator iter_type;
    explicit time_get_byname(const char* std_name, size_t refs = 0);
protected:
    virtual ~time_get_byname();
};

7.4.4.6 Time_put Members

The class time_put has one member function.

    iter_type put(iter_type s, ios_base& str,
                  char_type fill, const tm* t, const charT* pattern, const
                  charT* pat_end) const;
    iter_type put(iter_type s, ios_base& str,
                  char_type fill, const tm* t, char format,
                  char modifier = 0) const;

Remarks
Formats a localized time.
Returns an iterator immediately beyond the last character.

7.4.4.7 Time_put Virtual Functions

The class time_put has one virtual member function.

    iter_type do_put(iter_type s, ios_base&,
                    char_type fill, const tm* t, char format,
                    char modifier) const;

Remarks
Implements the public member function *put*.

### 7.4.4.8 Template Class Time_put_byname Synopsis

```cpp
namespace std {
    template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put_byname
    : public time_put<charT, OutputIterator>
    {
    public:
        typedef charT char_type;
        typedef OutputIterator iter_type;
        explicit time_put_byname(const char* std_name, size_t refs = 0);
    protected:
        virtual ~time_put_byname();
    };
}
```

#### 7.4.4.9 Extending The Behavior Of The Time Facets

The time facets can easily be extended and customized for many different cultures.

To stay portable one can derive from *time_get* and *time_put* and re-implement the behavior described above. Or one could take advantage of the EWL C++ implementation of these classes and build upon the existing functionality quite easily. Specifically you can easily alter the following data in the EWL time facets:

- The abbreviations of the weekday names
- The full weekday names
- The abbreviations of the month names
- The full month names
- The date-and-time format pattern (what `%c` will expand to)
- The date format pattern (what `%x` will expand to)
- The time format pattern (what `%X` will expand to)
- The 12 hour time format pattern (what `%r` will expand to)
- The strings used for AM/PM
- The strings used for the UTC offset
- The strings used for time zone names
- The default century to be used when parsing `%y`
7.4.4.10 Extending locale by using named locale facilities

The easiest way to specify the locale specific data is to use the named locale facilities.

When you create a named locale with a string that refers to a locale data file, the time facets parse that data file for time facet data.

```cpp
locale loc("my_locale");
```

The narrow file "my_locale" can hold time data for both narrow and wide time facets. Wide characters and strings can be represented in the narrow file using hexadecimal or universal format (e.g. `'\u06BD'`). Narrow time data starts with the keyword:

```
$time_narrow
```

And wide time data starts with the keyword:

```
$time_wide
```

Otherwise, the format for the time data is identical for the narrow and wide data.

There are twelve keywords that allow you to enter the time facet data:

1. abrev_weekday
2. weekday
3. abrev_monthname
4. monthname
5. date_time
6. am_pm
7. time_12hour
8. date
9. time
10. time_zone
11. utc_offset
12. default_century

You enter data with one of these keywords, followed by an equal sign '=' and then the data. You can specify any or all of the 12 keywords in any order. Data not specified will default to that of the "C" locale.

**NOTE**

See String Syntax for syntax details.
7.4.4.10.1 *abrev_weekday*

This keyword allows you to enter the abbreviations for the weekday names. There must be seven strings that follow this keyword, corresponding to Sun through Sat. The "C" designation is:

    abrev_weekday = Sun Mon Tue Wed Thu Fri Sat

7.4.4.10.2 *weekday*

This keyword allows you to enter the full weekday names. There must be seven strings that follow this keyword, corresponding to Sunday through Saturday. The "C" designation is:

    weekday = Sunday Monday Tuesday Wednesday Thursday Friday Saturday

7.4.4.10.3 *abrev_monthname*

This keyword allows you to enter the abbreviations for the month names. There must be twelve strings that follow this keyword, corresponding to Jan through Dec. The "C" designation is:

    abrev_monthname = Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

7.4.4.10.4 *monthname*

This keyword allows you to enter the full month names. There must be twelve strings that follow this keyword, corresponding to January through December. The "C" designation is:

    monthname =
    January February March April May June July August September October November December
7.4.4.10.5  date_time

This keyword allows you to enter the parsing/formatting string to be used when %c is encountered. The "C" locale has:

\[
\text{date_time} = \text{"%A %B %d %T %Y"}
\]

The date_time string must not contain %c, else an infinite recursion will occur.

7.4.4.10.6  am_pm

This keyword allows you to enter the two strings that designate AM and PM. The "C" locale specifies:

\[
\text{am_pm} = \text{am pm}
\]

7.4.4.10.7  time_12hour

This keyword allows you to enter the parsing/formatting string to be used when %r is encountered. The "C" locale has:

\[
\text{time_12hour} = \text{"%I:%M:%S %p"}
\]

The time_12hour string must not contain %r, else an infinite recursion will occur.

7.4.4.10.8  date

This keyword allows you to enter the parsing/formatting string to be used when %x is encountered. The "C" locale has:

\[
\text{date} = \text{"%A %B %d %Y"}
\]

The date string must not contain %x, else an infinite recursion will occur.

7.4.4.10.9  time
This keyword allows you to enter the parsing/formatting string to be used when \%X is encountered. The "C" locale has:

\[
time = \"%H:%M:%S\"
\]

The time string must not contain \%X, else an infinite recursion will occur.

### 7.4.4.10.10 \texttt{time\_zone}

This keyword allows you to enter two strings that designate the names of the locale's time zones: the first being the name for the time zone when Daylight Savings Time is not in effect, and the second name for when it is. The "C" locale has:

\[
time\_zone = \"\" \"
\]

This means that time zone information is not available in the "C" locale.

### 7.4.4.10.11 \texttt{utc\_offset}

This keyword allows you to enter two strings that designate the UTC offsets of the locale's time zones: the first being the offset for the time zone when Daylight Savings Time is not in effect, and the second string for when it is. The "C" locale has:

\[
utc\_offset = \"\" \"
\]

This means that UTC offset information is not available in the "C" locale.

### 7.4.4.10.12 \texttt{default\_century}

This keyword allows you to enter the default century which is used to create the correct year when parsing the \%y format. This format parses a number and then computes the year by adding it to 100*default\_century. The "C" locale has:

\[
default\_century = 20
\]

Assume a Date class. The I/O for the Date class can be written using time\_get and time\_put in a portable manner. The input operator might look like:

**Listing: Date Class Example Use**
The code extracts the time_get facet from the istream's locale and uses its get_date method to fill a tm. If the extraction was successful, then the data is transferred from the tm into the Date class.

Listing: The output method

```cpp
template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator <<(std::basic_ostream<charT, traits>& os, const Date& item)
{
    std::basic_ostream<charT, traits>::sentry ok(os);
    try
    {
        const std::time_get<charT>& tg =
            std::use_facet<std::time_get<charT> >(os.getloc());
        std::tm t;
        tg.get_date(os, 0, is, err, &t);
        if (!(err & std::ios_base::failbit))
            item = Date(t.tm_mon+1, t.tm_mday, t.tm_year+1900);
    }
    catch (...) 
    {
        err |= std::ios_base::badbit | std::ios_base::failbit;
    }
    os.setstate(err);
    return is;
}
```
if (ok)
{
    bool failed;
    try
    {
        const std::time_put<charT>& tp =
        std::use_facet<std::time_put<charT>>
        >(os.getloc());
        std::tm t;
        t.tm_mday = item.day();
        t.tm_mon = item.month() - 1;
        t.tm_year = item.year() - 1900;
        t.tm_wday = item.dayOfWeek();
        charT pattern[2] = {'%', 'x'};
        failed = tp.put(os, os, os.fill(), &t, pattern,
                        pattern+2).failed();
    }
    catch (...)
    {
        failed = true;
    }
    if (failed)
        os.setstate(std::ios_base::failbit |
                    std::ios_base::badbit);
}
return os;
}

After extracting the time_put facet from the ostream's locale, you transfer data from your
Date class into the tm (or the Date class could simply export a tm). Then the put method
is called with the tm and using the pattern "%x". There are several good things about the
Date's I/O methods:

• They are written in portable standard C++.
• They are culturally sensitive since they use the locale's time facets.
• They can handle narrow or wide streams.
• The streams can be in memory (e.g. stringstream) or file based streams (fstream)
For wide file streams, routing is automatically going through a codecvt that could (for example) be using something like UTF-8 to convert to/from the external file. They are relatively simple considering the tremendous flexibility involved.

With the Date's I/O done, the rest of the example is very easy. A French locale can be created with the following data in a file named "French":

```cpp
$time_narrow
date = "%A, le %d %B %Y"
weekday =
    dimanche lundi mardi mercredi jeudi vendredi samedi
abrev_weekday =
    dim lun mar mer jeu ven sam
monthname =
    janvier février mars avril mai juin juillet août
abrev_monthname =
    jan fév mar avr mai juin juil aoû sep oct nov déc
```

Now a program can read and write Date's in both English and French (and the Date class is completely ignorant of both languages).

**Listing: Example of dates in English and French**

```cpp
#include <locale>
#include <iostream>
#include <sstream>
#include "Date.h"

int main()
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale("French"));
    std::cout << "En Paris, c'est " << today << '\n';
    std::cout.imbue(std::locale("US"));
    std::cout << "But in New York it is " << today << '\n';
}
```

This program reads in a Date using the "C" locale from an istringstream. Then cout is imbued with "French" and the same Date is written out. And finally the same stream is imbued again with a "US" locale and the same Date is written out again. The output is:

En Paris, c'est samedi, le 24 février 2001
But in New York it is Saturday February 24 2001

For this example the "US" locale was implemented with an empty file. This was possible since the relevant parts of the "US" locale coincide with the "C" locale.
7.4.4.11 Extending by derivation

The behavior of the time facets can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons.

Naturally, you can derive from `time_get` and `time_put` and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the EWL C++ implementation if you wish (to make your job easier if portability is not a concern).

The central theme of the EWL time facets design is a non-standard facet class called `std::timepunct`:

**Listing: Template Class Timepunct Synopsis**

```cpp
template <class charT>
class timepunct
    : public locale::facet
{
public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;

    explicit timepunct(size_t refs = 0);

    const string_type& abrev_weekday(int wday) const
        {return __weekday_names_[7+wday];}
    const string_type& weekday(int wday) const
        {return __weekday_names_[wday];}
    const string_type& abrev_monthname(int mon) const
        {return __month_names_[12+mon];}
    const string_type& monthname(int mon) const
        {return __month_names_[mon];}
    const string_type& date_time() const
        {return __date_time_;}
    const string_type& am_pm(int hour) const
        {return __am_pm_[hour/12];}
```

---

EWL C++ Library Reference Manual, Rev. 10.x, 02/2014

Freescale Semiconductor, Inc.
const string_type& time_12hour() const
{    return __12hr_time_;}
const string_type& date() const
{return __date_;}
const string_type& time() const
{return __time_;}
const string_type& time_zone(int isdst) const
{return __time_zone_[isdst];}
const string_type& utc_offset(int isdst) const
{return __utc_offset_[bool(isdst)];}
int                default_century() const
{return __default_century_;}

static locale::id id;

protected:
    virtual ~timepunct() {} //

    string_type __weekday_names_[14];
    string_type __month_names_[24];
    string_type __am_pm_[2];
    string_type __date_time_;
    string_type __date_;        
    string_type __time_;        
    string_type __12hr_time_;   
    string_type __time_zone_[2];
    string_type __utc_offset_[2];
    int         __default_century_;
};

This class is analogous to numpunct and moneypunct. It holds all of the configurable data. The facets time_get and time_put refer to timepunct for the data and then behave accordingly. All of the data in timepunct is protected so that the constructor of a derived facet can set this data however it sees fit. The timepunct facet will set this data according to the "C" locale.
Both the full weekday names and the abbreviated weekday names are stored in __weekday_names__. The full names occupy the first seven elements of the array, and the abbreviated names get the last seven slots. Similarly for __month_names__.

The __am_pm__ member holds the strings that represent AM and PM, in that order.

The __date_time__ member holds the formatting/parsing string for the date-and-time. This is the member that gets queried when %c comes up. Do not put %c in this string or an infinite recursion will occur. The default for this string is "%A %B %d %T %Y".

The __date__ member holds the formatting/parsing string for the date. This is the member that gets queried when %x comes up. Do not put %x in this string or an infinite recursion will occur. The default for this string is "%A %B %d %Y".

The __time__ member holds the formatting/parsing string for the time. This is the member that gets queried when %x comes up. Do not put %x in this string or an infinite recursion will occur. The default for this string is "%H:%M:%S".

The __12hr_time__ member holds the formatting/parsing string for the 12-hour-time. This is the member that gets queried when %r comes up. Do not put %r in this string or an infinite recursion will occur. The default for this string is "%I:%M:%S %p".

The __time_zone__ member contains two strings. The first is the name of the time zone when Daylight Savings Time is not in effect. The second string is the name of the time zone when Daylight Savings Time is in effect. These can be used to parse or format the tm_isdst member of a tm. These strings may be empty (as they are in the "C" locale) which means that time zone information is not available.

The __utc_offset__ member contains two strings. The first represents the UTC offset when Daylight Savings Time is not in effect. The second string is the offset when Daylight Savings Time is in effect. These can be used to parse or format the tm_isdst member of a tm. These strings may be empty (as they are in the "C" locale) which means that UTC offset information is not available.

The final member, __default_century__ is an int representing the default century to assume when parsing a two digit year with %y. The value 19 represents the 1900’s, 20 represent’s the 2000’s, etc. The default is 20.

It is a simple matter to derive from timepunct and set these data members to whatever you see fit.

7.4.4.12  Timepunct_byname
You can use `timepunct_byname` to get the effects of a named locale for time facets instead of using a named locale.

The `time_get_byname` and `time_put_byname` facets do not add any functionality over `time_get` and `time_put`.

**Listing: Using Timepunct_byname**

```cpp
#include <locale>
#include <iostream>
#include <sstream>
#include "Date.h"

int main()
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale(std::locale(),
        new std::timepunct_byname<char>("French")));
    std::cout << "En Paris, c'est " << today << '\n';
    std::cout.imbue(std::locale(std::locale(),
        new std::timepunct_byname<char>("US")));
    std::cout << "But in New York it is " << today << '\n';
}
```

This has the exact same effect as the named locale example.

But the `timepunct_byname` example still uses the files "French" and "US". Below is an example timepunct derived class that avoids files but still captures the functionality of the above examples.

**Listing: Example Timepunct Facet Use**

```cpp
// The first job is to create a facet derived from timepunct
// that stores the desired data in the timepunct:

class FrenchTimepunct
    : public std::timepunct<char>
{
public:
    FrenchTimepunct();
```
FrenchTimepunct::FrenchTimepunct()
{
    __date__ = "%A, le %d %B %Y";
    __weekday_names_[0] = "dimanche";
    __weekday_names_[1] = "lundi";
    __weekday_names_[2] = "mardi";
    __weekday_names_[3] = "mercredi";
    __weekday_names_[4] = "jeudi";
    __weekday_names_[5] = "vendredi";
    __weekday_names_[6] = "samedi";
    __weekday_names_[7] = "dim";
    __weekday_names_[8] = "lun";
    __weekday_names_[9] = "mar";
    __weekday_names_[10] = "mer";
    __weekday_names_[11] = "jeu";
    __weekday_names_[12] = "ven";
    __weekday_names_[13] = "sam";
    __month_names_[0] = "janvier";
    __month_names_[1] = "février";
    __month_names_[2] = "mars";
    __month_names_[3] = "avril";
    __month_names_[4] = "mai";
    __month_names_[5] = "juin";
    __month_names_[6] = "juillet";
    __month_names_[7] = "août";
    __month_names_[8] = "septembre";
    __month_names_[9] = "octobre";
    __month_names_[10] = "novembre";
    __month_names_[11] = "décembre";
    __month_names_[12] = "jan";
    __month_names_[13] = "fév";
    __month_names_[14] = "mar";
    __month_names_[15] = "avr";
    __month_names_[16] = "mai";
//Though tedious, the job is quite simple.
//Next simply use your facet:
int main()
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale(std::locale(),
        new FrenchTimepunct));
    std::cout << "En Paris, c'est " << today << '\n';
    std::cout.imbue(std::locale::classic());
    std::cout << "But in New York it is " << today << '\n';
}

Here we have explicitly asked for the classic locale, instead of the "US" locale since the
two are the same (but executing classic() does not involve file I/O). Using the global
locale (locale()) instead of classic() would have been equally fine in this example.

7.4.5 The Monetary Category

There are five standard money classes.

- class money_base;
- template <class charT, class InputIterator = istreambuf_iterator<charT> > class
  money_get;
- template <class charT, class OutputIterator = ostreambuf_iterator<charT> > class
  money_put;
- template <class charT, bool International = false> class moneypunct;
- template <class charT, bool International = false> class moneypunct_byname;
The first of these (money_base) is not a facet, but the remaining four are. The money_base class is responsible only for specifying pattern components that will be used to specify how monetary values are parsed and formatted (currency symbol first or last, etc.).

The facets money_get and money_put are responsible for parsing and formatting respectively. Though their behavior is made up of virtual methods, and thus can be overridden via derivation, it will be exceedingly rare for you to feel the need to do so. Like the numeric facets, the real customization capability comes with the "punct" classes: moneypunct and moneypunct_byname.

A user-defined Money class (there will be an example later on) can use money_get and money_put in defining its I/O, and remain completely ignorant of whether it is dealing with francs or pounds. Instead clients of Money will imbue a stream with a locale that specifies this information. On I/O the facets money_get and money_put query moneypunct (or moneypunct_byname) for the appropriate locale-specific data. The Money class can remain blissfully ignorant of cultural specifics, and at the same time, serve all cultures!

### 7.4.5.1 A sample Money class

The very reason that we can design a Money class before we know the details of moneypunct customization is because the Money class can remain completely ignorant of this customization.

This Money class is meant only to demonstrate I/O. Therefore it is as simple as possible. We begin with a simple struct:

**Listing: A example demonstration of input and output**

```cpp
struct Money
{
    long double amount_;
};

// The I/O methods for this class follow a fairly standard formula,
// but reference the money facets to do the real work:
template<class charT, class traits>
std::basic_istream<charT, traits>&
operator >>(std::basic_istream<charT, traits>& is, Money& item)
{
    typename std::basic_istream<charT, traits>::sentry ok(is);
if (ok)
{
    std::ios_base::iostate err = std::ios_base::goodbit;
    try
    {
        const std::money_get<charT>& mg =
        std::use_facet<std::money_get<charT>> (is.getloc());
        mg.get(is, 0, false, is, err, item.amount_);
    }
    catch (...)
    {
        err |= std::ios_base::badbit | std::ios_base::failbit;
    }
    is.setstate(err);
}
return is;
}

template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator <<(std::basic_ostream<charT, traits>& os,
const Money& item)
{
    std::basic_ostream<charT, traits>::sentry ok(os);
    if (ok)
    {
        bool failed;
        try
        {
            const std::money_put<charT>& mp =
            std::use_facet<std::money_put<charT>> (os.getloc());
            failed = mp.put(os, false, os, os.fill(),
                            item.amount_).failed();
        }
        catch (...)
        {
        }
failed = true;
}
if (failed)
  os.setstate(std::ios_base::failbit |
    std::ios_base::badbit);
}
return os;
}
The extraction operator (\texttt{\textgreater\textgreater}) obtains a reference to \texttt{money\_get} from the stream's locale, and then simply uses its \texttt{get} method to parse directly into \texttt{Money}'s \texttt{amount\_}. The insertion operator (\texttt{\textless\textless}) does the same thing with \texttt{money\_put} and its put method. These methods are extremely flexible, as all of the formatting details (save one) are saved in the stream's locale. That one detail is whether we are dealing a local currency format, or an international currency format. The above methods hard wire this decision to "local" by specifying false in the \texttt{get} and \texttt{put} calls. The \texttt{moneypunct} facet can store data for both of these formats. An example difference between an international format and a local format is the currency symbol. The US local currency symbol is "$", but the international US currency symbol is "USD".

For completeness, we extend this example to allow client code to choose between local and international formats via a stream manipulator. See Matt Austern's excellent C/C++ Users Journal article: The Standard Librarian: User-Defined Format Flags for a complete discussion of the technique used here.

To support the manipulators, our simplistic \texttt{Money} struct is expanded in the following code example.

**Listing: Example of manipulator support.**

```c++
struct Money
{
  enum format {local, international};
  static void set_format(std::ios_base& s, format f)
    {flag(s) = f;}
  static format get_format(std::ios_base& s)
    {return static_cast<formatter>(flag(s));}
  static long& flag(std::ios_base& s);
  long double amount_;
};
```
An \texttt{enum} has been added to specify local or international format. But this \texttt{enum} is only defined within the \texttt{Money} class. There is no format data member within \texttt{Money}. That information will be stored in a stream by clients of \texttt{Money}. To aid in this effort, three static methods have been added: \texttt{set\_format}, \texttt{get\_format} and \texttt{flag}. The first two methods simply call \texttt{flag} which has the job of reading and writing the format information to the stream. Although \texttt{flag} is where the real work is going on, its definition is surprisingly simple.

\textbf{Listing: Money class flag}

```cpp
def Money::flag(std::ios_base& s)
{
    static int n = std::ios_base::xalloc();
    return s.iword(n);
}
```

As described in Austern's C/C++ User Journal article, \texttt{flag} uses the stream's \texttt{xalloc} facility to reserve an area of storage which will be the same location in all streams. And then it uses \texttt{iword} to obtain a reference to that storage for a particular stream. Now it is easier to see how \texttt{set\_format} and \texttt{get\_format} are simply writing and reading a long associated with the stream \texttt{s}.

To round out this manipulator facility we need the manipulators themselves to allow client code to write statements like:

```cpp
in >> international >> money;
out << local << money << '
';
```

These are easily accomplished with a pair of namespace scope methods:

\textbf{Listing: Money class manipulators}

```cpp
template<class charT, class traits>
std::basic_ios<charT, traits>&
local(std::basic_ios<charT, traits>& s)
{
    Money::set\_format(s, Money::local);
    return s;
}

template<class charT, class traits>
std::basic_ios<charT, traits>&
international(std::basic_ios<charT, traits>& s)
{
    Money::set\_format(s, Money::international);
    return s;
}
```
And finally, we need to modify the Money inserter and extractor methods to read this information out of the stream, instead of just blindly specifying false (local) in the get and put methods.

**Listing: Money class inserters and extractors**

```cpp
template<class charT, class traits>
std::basic_istream<charT, traits>&
operator >> (std::basic_istream<charT, traits>& is, Money& item)
{
    typename std::basic_istream<charT, traits>::sentry ok(is);
    if (ok)
    {
        std::ios_base::iostate err = std::ios_base::goodbit;
        try
        {
            const std::money_get<charT>& mg =
                std::use_facet<std::money_get<charT> >(is.getloc());
            mg.get(is, 0, Money::get_format(is) ==
                Money::international, is, err, item.amount_);
        } catch (...) {
            err |= std::ios_base::badbit |
                std::ios_base::failbit;
        }
        is.setstate(err);
    }
    return is;
}

template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator <<(std::basic_ostream<charT, traits>& os,
            const Money& item)
{
    std::basic_ostream<charT, traits>::sentry ok(os);
    if (ok)
    {
        bool failed;
        try
        {
            const std::money_put<charT>& mp =
                std::use_facet<std::money_put<charT> >(os.getloc());
            failed = mp.put(os, Money::get_format(os) ==
                Money::international, os, os.fill(),
                item.amount_).failed();
        }
        catch (...) {
```

(Continued in next frame)
```cpp
{  
    failed = true;
}

if (failed)
    os.setstate(std::ios_base::failbit | 
                std::ios_base::badbit);

return os;
}
```

Because we gave the enum `Money::local` the value 0, this has the effect of making local the default format for a stream.

We now have a simple `Money` class that is capable of culturally sensitive input and output, complete with local and international manipulators! To motivate the following sections on how to customize `moneypunct` data. Below is sample code that uses our `Money` class, along with the named locale facility:

**Listing: Example of using a money class**

```cpp
int main()
{
    std::istringstream in("USD (1,234,567.89)");

    Money money;
    in >> international >> money;

    std::cout << std::showbase << local << money << '
';
    std::cout << international << money << '
';
    std::cout.imbue(std::locale("Norwegian"));
    std::cout << local << money << '
';
    std::cout << international << money << '
';
}
```

And the output is:

```
$-1,234,567.89
USD (1,234,567.89)
-1 234 567,89 kr
NOK (1 234 567,89)
```

### 7.4.5.2 Template Class Money_get

The template class `money_get` is used for `locale` monetary input routines.

**Listing: Template Class Money_get Synopsis**
namespace std {

    template <class charT, 
    class InputIterator = istreambuf_iterator<charT> >
    class money_get : public locale::facet {
        public:
            typedef charT char_type;
            typedef InputIterator iter_type;
            typedef basic_string<charT> string_type;
            explicit money_get(size_t refs = 0);
            iter_type get(iter_type s, iter_type end, bool intl, 
                ios_base& f, ios_base::iostate& err, 
                long double& units) const;
            iter_type get(iter_type s, iter_type end, bool intl, 
                ios_base& f, ios_base::iostate& err, 
                string_type& digits) const;
            static locale::id id;
        protected:
            ~money_get(); //virtual
            virtual iter_type do_get(iter_type, iter_type, bool, ios_base&, 
                ios_base::iostate& err, long double& units) const;
            virtual iter_type do_get(iter_type, iter_type, bool, ios_base&, 
                ios_base::iostate& err, string_type& digits) const;
    }
}

7.4.5.2.1 Money_get Members

Localized member functions for inputting monetary values.

7.4.5.2.1.1 get

Inputs a localized monetary value.
Remarks

Returns an iterator immediately beyond the last character recognized as a valid monetary quantity.

7.4.5.2.1.2 Money_get Virtual Functions

Implementation functions for localization of the `money_get` public member functions.

```cpp
iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str, ios_base::iostate& err, long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str, ios_base::iostate& err, string_type& digits) const;
```

Remarks

Implements a localized monetary `get` function.

7.4.5.3 Template Class Money_put

The template class `money_put` is used for locale monetary output routines.

Listing: Template Class Money_put Synopsis

```cpp
namespace std {
    template <class charT,
        class OutputIterator = ostreambuf_iterator<charT> >
    class money_put : public locale::facet {
        public:
        typedef charT char_type;
        typedef OutputIterator iter_type;
        typedef basic_string<charT> string_type;
        explicit money_put(size_t refs = 0);
        iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, long double units) const;
        iter_type put(iter_type s, bool intl, ios_base& f,
```
char_type fill, const string_type& digits) const;
static locale::id id;
protected:
    -money_put(); //virtual
virtual iter_type
do_put(iter_type, bool, ios_base&, char_type fill,
long double units) const;
virtual iter_type
do_put(iter_type, bool, ios_base&, char_type fill,
const string_type& digits) const;
};

7.4.5.3.1 Money_put Members

Localized member functions for outputting monetary values.

7.4.5.3.1.1 put

Outputs a localized monetary value.

    iter_type put(iter_type s, bool intl, ios_base& f,
char_type fill, long double quant) const;
    iter_type put(iter_type s, bool intl, ios_base& f,
char_type fill, const string_type& quant) const;

Remarks

Returns an iterator immediately beyond the last character recognized as a valid monetary quantity.

7.4.5.3.1.2 Money_put Virtual Functions

Implementation functions for localization of the money_put public member functions.

    iter_type do_put(iter_type s, bool intl,
ios_base& str, char_type fill,
long double units) const;
    iter_type do_put(iter_type s, bool intl,
Remarks

Implements a localized put function.

7.4.5.4 Class Moneypunct

An object used for localization of monetary punctuation.

Listing: Template Class Moneypunct Synopsis

```cpp
namespace std {
    class money_base {
        public:
            enum part { none, space, symbol, sign, value }
            struct pattern { char field[4]; }
        
        template <class charT, bool International = false>
        class moneypunct : public locale::facet, public money_base {
            public:
                typedef charT char_type;
                typedef basic_string<charT> string_type;
                explicit moneypunct(size_t refs = 0);
                charT decimal_point() const;
                charT thousands_sep() const;
                string grouping() const;
                string_type curr_symbol() const;
                string_type positive_sign() const;
                string_type negative_sign() const;
                int frac_digits() const;
                pattern pos_format() const;
                pattern neg_format() const;
                static locale::id id;
                static const bool intl = International;
            protected:
```
Moneypunct Members

Member functions to determine the punctuation used for monetary formatting.

7.4.5.4.1.1 decimal_point

Determines what character to use as a decimal point.

    charT decimal_point() const;

Remarks

Returns a char to be used as a decimal point.

7.4.5.4.1.2 thousands_sep

Determines which character to use for a thousandths separator.

    charT thousands_sep() const;

Remarks

The character to be used for the thousands separator is specified with thousands_sep. Returns the character to use for a thousandths separator.
7.4.5.4.1.3  **grouping**

Determines a string that determines the grouping of thousands.

```cpp
string grouping() const;
```

**Remarks**

The grouping string specifies the number of digits to group, going from right to left. Returns the string that determines the grouping of thousands.

7.4.5.4.1.4  **curr_symbol**

Determines a string of the localized currency symbol.

```cpp
string_type curr_symbol() const;
```

**Remarks**

Returns the string of the localized currency symbol.

7.4.5.4.1.5  **positive_sign**

Determines a string of the localized positive sign.

```cpp
string_type positive_sign() const;
```

**Remarks**

Returns the string of the localized positive sign.

7.4.5.4.1.6  **negative_sign**

Determines a string of the localized negative sign.

```cpp
string_type negative_sign() const;
```

**Remarks**
Returns the string of the localized negative sign.

### 7.4.5.4.1.7 frac_digits

Determines a string of the localized fractional digits.

```cpp
int frac_digits() const;
```

**Remarks**

Returns the string of the localized fractional digits.

### 7.4.5.4.1.8 pos_format

Determines the format of the localized non-negative values.

```cpp
pattern pos_format() const;
```

**Remarks**

These keywords allow you to enter the format for both positive and negative values. There are 5 keywords to specify a format:

- none
- space
- symbol
- sign
- value

A monetary format is a sequence of four of these keywords. Each value: symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last. The behavior of breaking any of these rules is undefined. The default pattern for positive values, and for local and international formats is:

```cpp
pos_format = symbol sign none value
```

Returns the pattern initialized to a positive value.
7.4.5.4.1.9  neg_format

Determines the format of the localized non-negative values.

    pattern neg_format() const;

Remarks

These keywords allow you to enter the format for both positive and negative values. There are 5 keywords to specify a format:

- none
- space
- symbol
- sign
- value

A monetary format is a sequence of four of these keywords. Each value: symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last. The behavior of breaking any of these rules is undefined. The default pattern for negative values, and for local and international formats is:

    neg_format = symbol sign none value

Returns the pattern initialized to a negative value.

7.4.5.4.1.10  Moneypunct Virtual Functions

Virtual functions that implement the localized public member functions.

    charT do_decimal_point() const;

Implements decimal_point.

    charT do_thousands_sep() const;

Implements thousands_sep.

    string do_grouping() const;
Implements `grouping`.

```cpp
string_type do_curr_symbol() const;
```

Implements `cur_symbol`.

```cpp
string_type do_positive_sign() const;
```

Implements `positive_sign`.

```cpp
string_type do_negative_sign() const;
```

Returns the string to use to indicate a negative value.

```cpp
int do_frac_digits() const;
```

Implements `frac_digits`.

```cpp
pattern do_pos_format() const;
```

Implements `pos_format`.

```cpp
pattern do_neg_format() const;
```

Implements `neg_format`.

### 7.4.5.5 Extending `moneypunct` by derivation

It is easy enough to derive from `moneypunct` and override the virtual functions in a portable manner.

But `moneypunct` also has a non-standard protected interface that you can take advantage of if you wish. There are nine protected data members:

```cpp
charT __decimal_point_;  
charT __thousands_sep_;  
string __grouping_;  
string_type __cur_symbol_;  
string_type __positive_sign_;  
string_type __negative_sign_;  
int __frac_digits_;  
pattern __pos_format_;  
pattern __neg_format_;  
```
A derived class could set these data members in its constructor to whatever is appropriate, and thus not need to override the virtual methods.

**Listing: Extending Moneypunct by derivation**

```cpp
struct mypunct
    : public std::moneypunct<char, false>
{
    mypunct();
};

mypunct::mypunct()
{
    __decimal_point_ = ',';
    __thousands_sep_ = ' ';
    __cur_symbol_ = "kr";
    __pos_format_.field[0] = __neg_format_.field[0] = char(sign);
    __pos_format_.field[1] = __neg_format_.field[1] = char(value);
    __pos_format_.field[2] = __neg_format_.field[2] = char(space);
    __pos_format_.field[3] = __neg_format_.field[3] = char(symbol);
}
```

```cpp
int
main()
{
    std::locale loc(std::locale(), new mypunct);
    std::cout.imbue(loc);
    // ...
}
```

Indeed, this is just what `moneypunct_byname` does after reading the appropriate data from a locale data file.

### 7.4.5.6 Template Class Moneypunct_byname

A template class for implementation of the `moneypunct` template class.

**Listing: Template Class Moneypunct_byname Synopsis**

```cpp
namespace std {
    template <class charT, bool Intl = false>
    class moneypunct_byname : public moneypunct<charT, Intl> {
        public:
```
typedef money_base::pattern pattern;
typedef basic_string<charT> string_type;
explicit moneypunct_byname(const char*, size_t refs = 0);
protected:
- moneypunct_byname(); // virtual
virtual charT do_decimal_point() const;
virtual charT do_thousands_sep() const;
virtual string do_grouping() const;
virtual string_type do_curr_symbol() const;
virtual string_type do_positive_sign() const;
virtual string_type do_negative_sign() const;
virtual int do_frac_digits() const;
virtual pattern do_pos_format() const;
virtual pattern do_neg_format() const;
};
}

When a named locale is created:

    std::locale my_loc("MyLocale");

this places the facet moneypunct_byname("MyLocale") in the locale. The moneypunct_byname
constructor considers the name it is constructed with as the name of a data file which may
or may not contain moneypunct data. There are 4 keywords that mark the beginning of
moneypunct data in a locale data file.

- $money_local_narrow
- $money_international_narrow
- $money_local_wide
- $money_international_wide

These data sections can appear in any order in the locale data file. And they are all
optional. Any data not specified defaults to that of the "C" locale. Wide characters and
strings can be represented in the narrow locale data file using hexadecimal or universal
format (for example, \\u06bd). See the rules for Strings and Characters in Locale Data
Files for more syntax details.
7.4.5.7  Data file syntax

The syntax for entering `moneypunct` data is the same under all four keywords. There are 9 keywords that can be used within a `$money_XXX` data section to specify `moneypunct` data. The keywords can appear in any order and they are all optional.

- `decimal_point`
- `thousands_sep`
- `grouping`
- `curr_symbol`
- `positive_sign`
- `negative_sign`
- `frac_digits`
- `pos_format`
- `neg_format`

Each of these keywords is followed by an equal sign (=) and then the appropriate data (described below).

7.4.5.7.1  `decimal_point`

The decimal point data is a single character, as in:

```
decimal_point = '.'
```

**Remarks**

The default decimal point is '.'

7.4.5.7.2  `thousands_sep`

The character to be used for the thousands separator is specified with `thousands_sep`, as in:

```
thousands_sep = ','
```

**Remarks**

The default thousands separator is ',.'
7.4.5.7.3 grouping

The grouping string specifies the number of digits to group, going from right to left.

Remarks

For example, the grouping: 321 means that the number 12345789 would be printed as in:

1,2,3,4,5,6,7,89

The above grouping string can be specified as:

    grouping = 321

A grouping string of "0" or "" means: don't group. The default grouping string is "3".

7.4.5.7.4 curr_symbol

The currency symbol is specified as a string by curr_symbol, as in:

    curr_symbol = $

It is customary for international currency symbols to be four characters long, but this is not enforced by the locale facility. The default local currency symbols is "$". The default international currency symbol is "$USD".

7.4.5.7.5 positive_sign

The string to be used for the positive sign is specified by positive_sign. Many locales set this as the empty string, as in:

    positive_sign = ""

Remarks

The default positive sign is the empty string.

7.4.5.7.6 negative_sign

The negative sign data is a string specified by negative_sign, as in:
negative_sign = ()

**Remarks**

The precise rules for how to treat signs that are longer than one character are laid out in the standard. Suffice it to say that this will typically enclose a negative value in parentheses.

The default negative sign for local formats is "-", and for international formats is "()".

### 7.4.5.7.7 frac_digits

The number of digits to appear after the decimal point is specified by `frac_digits`, as in:

```
frac_digits = 2
```

**Remarks**

The default value is 2.

### 7.4.5.7.8 pos_format / neg_format

These keywords allow you to enter the format for both positive and negative values.

**Remarks**

There are 5 keywords to specify a format:

- none
- space
- symbol
- sign
- value

A monetary format is a sequence of four of these keywords. Each value: `symbol`, `sign`, `value`, and either `space` or `none` appears exactly once. The value `none`, if present, is not first; the value `space`, if present, is neither first nor last. The behavior of breaking any of these rules is undefined.

The default pattern for positive and negative values, and for local and international formats is:
pos_format = symbol sign none value

neg_format = symbol sign none value

Notice that in the following listing not all of the fields have been specified because the default values for these fields were already correct. On the other hand, it does not hurt to specify default data to improve (human) readability in the data file.

**Listing: Example Data file**

To have the example code run correctly, we need a file named "Norwegian" containing the following data:

```cpp
$money_local_narrow

decimal_point = ',',
thousands_sep = ' ',
curr_symbol = kr
pos_format = sign value space symbol
neg_format = sign value space symbol

$money_international_narrow

decimal_point = ',',
thousands_sep = ' ',
curr_symbol = "NOK "
```

### 7.4.6 The Message Retrieval Category

The messages facet is the least specified facet in the C++ standard. Just about everything having to do with messages is implementation defined.

**Listing: Template Class Messages Synopsis**

```cpp
namespace std {
    class messages_base {
        public:
            typedef int catalog;
    };
    template <class charT>
    class messages
        : public locale::facet,
          public messages_base
```
The intent is that you can use this class to read messages from a catalog. There may be multiple sets of messages in a catalog. And each message set can have any number of int/string pairs. But beyond that, the standard is quiet.

Does the string fn in open refer to a file? If so, what is the format of the set/msgid/string data to be read in from the file? There is also a messages_byname class that derives from messages. What functionality does messages_byname add over messages?

Unfortunately the answers to all of these questions are implementation defined. This document seeks to answer those questions. Please remember that applications depending on these answers will probably not be portable to other implementations of the standard C++ library.

### 7.4.6.1 Messages Members

Public member functions for catalog message retrieval.
7.4.6.1.1  open

Opens a message catalog for reading

```cpp
catalog open(const basic_string<char>& name, const locale& loc) const;
```

Remarks
Returns a value that may be passed to get to retrieve a message from a message catalog.

7.4.6.1.2  get

Retrieves a message from a message catalog.

```cpp
string_type get(catalog cat, int set, int msgid, const string_type& dfault) const;
```

Remarks
Returns the message in the form of a string.

7.4.6.1.3  close

Closes a message catalog.

```cpp
void close(catalog cat) const;
```

7.4.6.1.4  Messages Virtual Functions

Virtual functions used to localize the public member functions.

```cpp
catalog do_open(const basic_string<char>& name, const locale& loc) const;
```

Implements open.

```cpp
string_type do_get(catalog cat, int set, int msgid, const string_type& dfault) const;
```
Implements `get`.

```cpp
void do_close(catalog cat) const;
```

Implements `close`.

### 7.4.6.2 EWL C++ implementation of messages

The Embedded Warrior Library for C++ has a custom implementation of messages.

```cpp
Example code to open a catalog:
typedef std::messages<char> Msg;
const Msg& ct = std::use_facet<Msg>(std::locale::classic());
Msg::catalog cat = ct.open("my_messages",
   std::locale::classic());
if (cat < 0)
{
   std::cout << "Can't open message file\n";
   std::exit(1);
}
```

The first line simply type defines `messages<char>` for easier reading or typing. The second line extracts the messages facet from the "C" locale. The third line instructs the messages facet to look for a file named "my_messages" and read message set data out of it using the classic ("C") locale (one could specify a locale with a specialized codecvt facet for reading the data file). If the file is not found, the open method returns -1. The facet `messages<char>` reads data from a narrow file (`ifstream`). The facet `messages<wchar_t>` reads data from a wide file (`wifstream`).

The messages data file can contain zero or more message data sets of the format:

- `$set setid`
- `msgid message`
- `msgid message`
- `msgid message`
- ...

The keyword `$set` begins a message data set. The `setid` is the set number. It can be any int. Set id's do not need to be contiguous. But the set id must be unique among the sets in this catalog.

The `msgid` is the message id number. It can be any int. Message id's do not need to be contiguous. But the message id must be unique among the messages in this set.
The message is an optionally quoted (") string that is the message for this setid and msgid. If the message contains white space, it must be quoted. The message can have characters represented escape sequences using the hexadecimal or universal format. For example (see also String Syntax):

"\u0048\u0069\u0020\u0054\u0068\u0065\u0072\u0065\u0021"

The message data set terminates when the data is not of the form

msgid message

Thus, there are no syntax errors in this data. Instead, a syntax error is simply interpreted as the end of the data set. The catalog file can contain data other than message data sets. The messages facet will scan the file until it encounters $set setid.

Listing: Example of message facet

An example message data file might contain:
$set 1
1 "First Message"
2 "Error in foo"
3 Baboo
4 "\u0048\u0069\u0020\u0054\u0068\u0065\u0072\u0065\u0021"
$set 2
1 Ok
2 Cancel

A program that uses messages to read and output this file follows:

```cpp
#include <locale>
#include <iostream>
int main() {
    typedef std::messages<char> Msg;
    const Msg& ct = std::use_facet<Msg>(std::locale::classic());
    Msg::catalog cat = ct.open("my_messages",
                               std::locale::classic());
    if (cat < 0) {
        std::cout << "Can't open message file\n";
        return 1;
    }
    std::string eof("no more messages");
    for (int set = 1; set <= 2; ++set) {
        std::cout << "set " << set << "\n";
        for (int msgid = 1; msgid < 10; ++msgid) {
            std::string msg = ct.get(cat, set, msgid, eof);
            if (msg == eof)
```
The output of this program is:

set 1

1   First Message
2   Error in foo
3   Baboo
4   Hi There!

set 2

1   Ok
2   Cancel

7.4.6.3 Template Class Messages_byname Synopsis

The class `messages_byname` adds no functionality over messages.

The `const char*` that it is constructed with is ignored. To localize `messages` for a specific culture, either open a different catalog (file), or have different sets in a catalog represent `messages` for different cultures.

Listing: Template Class Messages_byname Synopsis

```cpp
namespace std {
    template <class charT>
    class messages_byname : public messages<charT> {
    public:
        typedef messages_base::catalog catalog;
        typedef basic_string<charT> string_type;
        explicit messages_byname(const char*, size_t refs = 0);
    protected:
    ~messages_byname(); // virtual

    virtual catalog do_open(const basic_string<char>&, const locale&) const;

    virtual string_type do_get(catalog, int set, int msgid,
```
7.4.6.4 Extending messages by derivation

If you are on a platform without file support, or you do not want to use files for messages for other reasons, you may derive from messages and override the virtual methods as described by the standard.

Additionally you can take advantage of the EWL C++ specific protected interface of messages if you wish (to make your job easier if portability is not a concern).

The messages facet has the non-virtual protected member:

```cpp
string_type& __set(catalog c, int set, intmsgid);
```

You can use this to place the quadruple (c, set,msgid, string) into messages' database. The constructor of the derived facet can fill the database using multiple calls to __set.

Below is an example of such a class. This example also overrides do_open to double check that the catalog name is a valid name, and then return the proper catalog number. And do_close is also overridden to do nothing. The messages destructor will reclaim all of the memory used by its database:

The main program (client code) in the Example of extending message by derivation is nearly identical to the previous example. Here we simply create and use the customized messages facet. Alternatively we could have created a locale and installed this facet into it. And then extracted the facet back out of the locale using use_facet as in the first example.

Listing: Example of extending message by derivation

```cpp
#include <locale>
#include <iostream>
#include <string>
#include <map>

class MyMessages
    : public std::messages<char>
{
public:
    MyMessages();
}
```
protected:
    virtual catalog do_open(const std::string& fn,
        const std::locale&) const;
    virtual void do_close(catalog) const{}

private:
    std::map<std::string, catalog> catalogs_;
return 1;
}
std::string eof("no more messages");
for (int set = 1; set <= 2; ++set)
{
    std::cout << "set " << set << "\n"
for (int msgid = 1; msgid < 10; ++msgid)
{
    std::string msg = ct.get(cat, set, msgid, eof);
    if (msg == eof)
        break;
    std::cout << msgid << "\t" << msg << '\n';
}
std::cout << '\n';
}
ct.close(cat);
}

The output of this program is:

set 1

1 set 1: first message
2 set 1: second message
3 set 1: third message

set 2

1 set 2: first message
2 set 2: second message
3 set 2: third message

7.4.7 Program-defined Facets

A C++ program may add its own locales to be added to and used the same as the built in facets.

To do this derive a class from locale::facet with the static member static locale::id.id.
The C++ header `<clocale>` are the same as the C header locale but in standard namespace.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>LC_ALL</td>
<td>LC_CTYPE</td>
</tr>
<tr>
<td>Macro</td>
<td>LC_COLLATE</td>
<td>LC_MONETARY</td>
</tr>
<tr>
<td>Macro</td>
<td>LC_NUMERIC</td>
<td>LC_TIME</td>
</tr>
<tr>
<td>Macro</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>Struct</td>
<td>lconv</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>localeconv</td>
<td>setlocale</td>
</tr>
</tbody>
</table>
Chapter 8
Containers Library

Containers are used to store and manipulate collections of information.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Container Requirements
- Sequences
- Associative Containers
- Template Class Bitset

8.1 Container Requirements

Container objects store other objects and control the allocation and de-allocation of those objects.

8.1.1 All containers must meet basic requirements.

The swap(), equal() and lexicographical_compare() algorithms are defined in the algorithm library for more information see Algorithms Library.

The member function size() returns the number of elements in a container.

The member function begin() returns an iterator to the first element and end returns an iterator to the last element.

If begin() equals end() the container is empty.

Copy constructors for container types copy and allocator argument from their first parameter. All other constructors take an Allocator reference argument.
The member function `get_allocator()` returns a copy of the Allocator object used in construction of the container.

If an iterator type of the container is bi-directional or a random access iterator the container is reversible.

### 8.1.2 Unless specified containers meet these requirements.

If an exception is thrown by an insert() function while inserting a single element, that function has no effects.

If an exception is thrown by a push_back() or push_front() function, that function has no effects.

The member functions erase(), pop_back() or pop_front() do not throw an exception.

None of the copy constructors or assignment operators of a returned iterator throw an exception.

The member function swap() does not throw an exception. Except if an exception is thrown by the copy constructor or assignment operator of the container's compare object.

The member function swap() does not invalidate any references, pointers, or iterators referring to the elements of the containers being swapped.

### 8.1.3 Sequences Requirements

A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement.

The Library includes three kinds of sequence containers `vector`, `lists`, `deque` and `adaptors` classes.

### 8.1.3.1 Additional Requirements

The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased.

If no prior element exists for `a.erase` then `a.end()` is returned.

- The previous conditions are true for `a.erase(q1,q2)` as well.
For every sequence defined in this clause the constructor

```cpp
template <class InputIterator>
X(InputIterator f, InputIterator l,
   const Allocator& a = Allocator())
```

- shall have the same effect as:

```
X(static_cast<typename X::size_type>(f),
   static_cast<typename X::value_type>(l),a)
```

- if InputIterator is an integral type.

Member functions in the forms:

```cpp
template <class InputIterator>
rt fx1(iterator p, InputIterator f, InputIterator l);
template <class InputIterator>
rt fx2(InputIterator f, InputIterator l);
template <class InputIterator>
rt fx3(iterator i1, iteraror i2, InputIterator f, InputIterator l);
```

- shall have the same effect, respectively, as:

```
fx1(p, static_cast<typename X::size_type>(f),
    static_cast<typename X::value_type>(l));
fx2(static_cast<typename X::size_type>(f),
    static_cast<typename X::value_type>(l));
fx3(i1, i2, static_cast<typename X::size_type>(f),
    static_cast<typename X::value_type>(l));
```

- if InputIterator is an integral type.

The member function `at()` provides bounds-checked access to container elements.

The member function `at()` throws `out_of_range` if `n >= a.size()`.

### 8.1.4 Associative Containers Requirements

Associative containers provide an ability for optimized retrieval of data based on keys.

Associative container are parameterized on Key and an ordering relation. Furthermore, map and multimap associate an arbitrary type T with the key.

The phrase "equivalence of keys" means the equivalence relation imposed by the comparison and not the `operator ==` on keys.

An associative container supports both unique keys as well as support fir equivalent keys.
• The classes set and map support unique keys.
• The classes multiset and multimap support equivalent keys.

An iterator of an associative container must be of the bidirectional iterator category.
The insert members shall not affect the validity of iterators.
Iterators of associative containers iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them.

8.2 Sequences

The sequence libraries consist of several headers.

• Template Class Deque
• Template Class List
• Container Adaptors
• Template Class Queue
• Template Class Priority_queue
• Template Class Stack
• Template Class Vector
• Class Vector<bool>

8.2.1 Template Class Deque

A deque is a kind of sequence that supports random access iterators.
The deque class also supports insert and erase operations at the beginning middle or the end. However, deque is especially optimized for pushing and popping elements at the beginning and end.
A deque satisfies all of the requirements of a container and of a reversible container as well as of a sequence.

8.2.1.1 Constructors
The deque constructor creates an object of the class deque.

```
explicit deque(const Allocator& = Allocator());
explicit deque(size_type n, const T& value = T(),
const Allocator& = Allocator());
template <class InputIterator>
deque(InputIterator first, InputIterator last,
const Allocator& = Allocator());
```

### 8.2.1.2 assign

The assign function is overloaded to allow various types to be assigned to a deque.

```
template <class InputIterator>
void assign (InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
```

**DequeCapacity**

The class deque has one member function to resize the deque.

### 8.2.1.3 resize

This function resizes the deque.

```
void resize(size_type sz, T c = T());
```

**DequeModifiers**

The deque class has member functions to modify the deque.

### 8.2.1.4 insert

The insert function is overloaded to insert a value into deque.

```
iterator insert(iterator position, const T& x);
void insert
(iterator position, size_type n, const T& x);
template <class InputIterator>
void insert
```
8.2.1.5 erase

An overloaded function that allows the removal of a value at a position.

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

Remarks

An iterator to the position erased.

DequeSpecializedAlgorithms

Deque has one specialize swap function.

8.2.1.6 swap

Swaps the element at one position with another.

```
template <class T, class Allocator>
void swap (deque<T,Allocator>& x, deque<T,Allocator>& y);
```

8.2.2 Template Class List

A list is a sequence that supports bidirectional iterators and allows insert and erase operations anywhere within the sequence.

In a list fast random access to list elements is not supported.

A list satisfies all of the requirements of a container as well as those of a reversible container and of a sequence except for operator[] and the member function at which are not included.
8.2.2.1 Constructors

The overloaded list constructors create objects of type list.

```cpp
explicit list(const Allocator& = Allocator());
explicit list(size_type n, const T& value = T(),
             const Allocator& = Allocator());
template <class InputIterator>
list(InputIterator first, InputIterator last,
     const Allocator& = Allocator());
```

8.2.2.2 assign

The overloaded assign function allows values to be assigned to a list after construction.

```cpp
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
```

ListCapacity

The list class provides for one member function to resize the list.

8.2.2.3 resize

Resizes the list.

```cpp
void resize(size_type sz, T c = T());
```

ListModifiers

The list class has several overloaded functions to allow modification of the list object.

8.2.2.4 insert

The insert member function insert a value at a position.

```cpp
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
```
8.2.2.5 push_front

The push_front member function pushes a value at the front of the list.

```cpp
void push_front(const T& x);
```

8.2.2.6 push_back

The push_back member function pushes a value onto the end of the list.

```cpp
void push_back(const T& x);
```

8.2.2.7 erase

The erase member function removes a value at a position or range.

```cpp
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

Remarks

Returns an iterator to the last position.

8.2.2.8 pop_front

The pop_front member function removes a value from the top of the list.

```cpp
void pop_front();
```
### 8.2.2.9 pop_back

The `pop_back` member function removes a value from the end of the list.

```cpp
void pop_back();
```

### 8.2.2.10 clear

Clears a list by removing all elements.

```cpp
void clear();
```

### ListOperations

The list class provides for operations to manipulate the list.

### 8.2.2.11 splice

Moves an element or a range of elements in front of a position specified.

```cpp
void splice
(iterator position, list<T,Allocator>& x);
void splice
(iterator position, list<T,Allocator>& x, iterator i);
void splice
(iterator position, list<T,Allocator>& x, iterator first, iterator last);
```

### 8.2.2.12 remove

Removes all element with a value.

```cpp
void remove(const T& value);
```
8.2.2.13 remove_if

Removes all element for which the predicate is true.

```
template <class Predicate>
void remove_if(Predicate pred);
```

8.2.2.14 unique

Removes duplicates of consecutive elements.

```
void unique();
template <class BinaryPredicate>
void unique(BinaryPredicate binary_pred);
```

8.2.2.15 merge

Moves sorted elements into a list according to the compare argument.

```
void merge(list<T,Allocator>& x);
template <class Compare>
void merge(list<T,Allocator>& x, Compare comp);
```

8.2.2.16 reverse

Reverses the order of the list.

```
void reverse();
```

8.2.2.17 sort

Sorts a list according to the Compare function or by less than value for the parameterless version.
ListSpecializedAlgorithms
The list class provides a swapping function.

8.2.2.18 swap
Changes the position of the first argument with the second argument.

    template <class T, class Allocator>
    void swap (list<T,Allocator>& x, list<T,Allocator>& y);

8.2.3 Container Adaptors
Container adaptors take a Container template parameter so that the container is copied into the Container member of each adaptor.

8.2.4 Template Class Queue
Any of the sequence types supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue.

8.2.4.1 operator ==
A user supplied operator for the queue class that compares the queue's data member.

    bool operator ==

Remarks
Returns true if the data members are equal.
8.2.4.2 operator <

A user supplied operator for the queue class that compares the queue's data member.

    bool operator <

Remarks
Returns true if the data member is less than the compared queue.

8.2.5 Template Class Priority_queue

You can instantiate any `priority_queue` with any sequence that has random access iterator and supporting operations `front()`, `push_back()` and `pop_back()`.

Instantiation of a priority_queue requires supplying a function or function object for making the priority comparisons.

8.2.5.1 Constructors

Creates an object of type `priority_queue`.

    priority_queue(const Compare& x = Compare(),
                   const Container& y = Container());
    template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last,
                   const Compare& x = Compare(),
                   const Container& y = Container());

priority_queue members

The class priority_queue provides public member functions for manipulation the priority_queue.

8.2.5.2 push
Inserts an element into the priority_queue.

```cpp
void push(const value_type& x);
```

### 8.2.5.3 pop

Removes an element from a priority_queue.

```cpp
void pop();
```

### 8.2.6 Template Class Stack

A stack class may be instantiated by any sequence supporting operations `back()`, `push_back()` and `pop_back()`.

#### 8.2.6.1 Public Member Functions

This section describes public member functions.

#### 8.2.6.1.1 Constructors

Creates an object of type stack with a container object.

```cpp
explicit stack(const Container& = Container());
```

#### 8.2.6.1.2 empty

Signifies when the stack is empty

```cpp
bool empty() const;
```

**Remarks**

Returns true if there are no elements in the stack.
8.2.6.1.3  size

Gives the number of elements in a stack.

    size_type size() const;

Remarks
Returns the number of elements in a stack.

8.2.6.1.4  top

Gives the top element in the stack.

    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }

Remarks
Returns the value at the top of the stack.

8.2.6.1.5  push

Puts a value onto a stack.

    void push(const value_type& x) { c.push_back(x); }

8.2.6.1.6  pop

Removes an element from a stack.

    void pop();

8.2.7  Template Class Vector
A vector is a kind of sequence container that supports random access iterators.

You can use insert and erase operations at the end and in the middle but at the end is faster.

A vector satisfies all of the requirements of a container and of a reversible container and of a sequence. It also satisfies most of the optional sequence requirements with the exceptions being push_front and pop_front member functions.

### 8.2.7.1 Constructors

The vector class provides overloaded constructors for creation of a vector object.

```cpp
vector(const Allocator& = Allocator());
explicit vector (size_type n, const T& value = T(),
    const Allocator& = Allocator());
template <class InputIterator>
vector(InputIterator first, InputIterator last,
    const Allocator& = Allocator());
vector(const vector<T,Allocator>& x);
```

### 8.2.7.2 assign

The member function assign allows you to assign values to an already created object.

```cpp
template <class InputIterator>
void assign
    (InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
```

### 8.2.7.3 capacity

Tells the maximum number of elements the vector can hold.

```cpp
size_type capacity() const;
```

**Remarks**

Returns the maximum number of elements the vector can hold.
8.2.7.4 resize

Resizes a vector if a second argument is given, the elements are filled with that value.

```cpp
void resize(size_type sz, T c = T());
```

VectorModifiers

The vector class provides various member functions for vector data manipulation.

8.2.7.5 insert

The member function insert inserts a value or a range of values at a set position.

```cpp
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
  void insert(iterator position, InputIterator first, InputIterator last);
```

8.2.7.6 erase

Removes elements at a position or for a range.

```cpp
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

VectorSpecializedAlgorithms

The vector class provides for a specialized swap function.

8.2.7.7 swap

Swaps the data of one argument with the other argument.

```cpp
template <class T, class Allocator> void swap
  (vector<T,Allocator>& x, vector<T,Allocator>& y);
```
8.2.8 Class Vector<bool>

A specialized vector for bool elements is provided to optimize allocated space.

A EWL bitvector class is available for efficient bool vector manipulations. Refer to Bitvector Class Library for more information.

8.3 Associative Containers

The associative container library consists of four template container classes.

- Template Class Map
- Template Class Multimap
- Template Class Set
- Template Class Multiset

8.3.1 Template Class Map

The map class is an associative container that supports unique keys and provides for retrieval of values of another type \( T \) based on the keys.

The map template class supports bidirectional iterators.

The template class map satisfies all of the requirements of a normal container and those of a reversible container, as well as an associative container.

A map also provides operations for unique keys.

8.3.1.1 Constructors

The map class provides an overloaded constructor for creating an object of type map.

```c++
explicit map(const Compare& comp = Compare(),
const Allocator& = Allocator());
template <class InputIterator> map (InputIterator first,
```
8.3.1.2 Map Element Access

The map class includes an element access operator.

8.3.1.2.1 operator []

Access an indexed element.

\[ T& \text{ operator[]} (\text{const key_type}& \ x); \]

Remarks

Returns the value at the position indicated.

8.3.1.3 Map Operations

The map class includes member functions for map operations.

8.3.1.3.1 find

Finds an element based upon a key.

\[ \text{iterator find(\text{const key_type}& \ x);} \]
\[ \text{const_iterator find(\text{const key_type}& \ x) \ const;} \]

Remarks

Returns the position where the element is found.

8.3.1.3.2 lower_bound

Finds the first position where an element based upon a key would be inserted.
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

Remarks
Returns the first position where an element would be inserted.

8.3.1.3.3 upper_bound

Finds the last position where an element based upon a key would be inserted.

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type &x) const;

Remarks
Returns the last position where an element would be inserted.

8.3.1.3.4 equal_range

Finds both the first and last position in a range where an element based upon a key would be inserted.

pair<iterator, iterator> equal_range (const_key_type &x);
pair<const_iterator, const_iterator> equal_range
(const key_type& x) const;

Remarks
Returns a pair of elements representing a range for insertion.

8.3.1.4 Map Specialized Algorithms

The map class provides for a method to swap elements.

8.3.1.4.1 swap

Swaps the first argument with the second argument.

template <class Key, class T, class Compare, class Allocator>
void swap
8.3.2 Template Class Multimap

A multimap container supports equivalent keys that may contain multiple copies of the same key value. Multimap provides for fast retrieval of values of another type based on the keys.

Multimap supports bidirectional iterators.

The multimap satisfies all of the requirements of a container, reversible container and associative containers.

Multimap supports the \texttt{a\_eq} operations but not the \texttt{a\_uniq} operations.

For a \texttt{multimap<Key,T>} the \texttt{key\_type} is \texttt{Key} and the \texttt{value\_type} is \texttt{pair<const Key,T>}

8.3.2.1 Constructors

The multimap constructor is overloaded for creation of a multimap object.

\begin{verbatim}
explicit multimap
 (const Compare& comp = Compare(),
  const Allocator& = Allocator());
template <class InputIterator>
multimap
 (InputIterator first, InputIterator last,
  const Compare& comp = Compare(),
  const Allocator& = Allocator());
\end{verbatim}

8.3.2.2 Multimap Operations

The multimap class includes member functions for manipulation of multimap data.

8.3.2.2.1 \texttt{find}

Finds a value based upon a key argument.
iterator find(const key_type &x);
const_iterator find(const key_type& x) const;

Remarks
Returns the position where the element is at.

8.3.2.2.2 lower_bound

Finds the first position where an element based upon a key would be inserted.

iterator lower_bound (const key_type& x);
const_iterator lower_bound (const key_type& x) const;

Remarks
Returns the position where an element was found.

8.3.2.2.3 equal_range

Finds the first and last positions where a range of elements based upon a key would be inserted.

pair<iterator, iterator> equal_range
(const key_type& x);
pair<const_iterator, const_iterator> equal_range
(const_key_type& x) const;

Remarks
Returns a pair object that represents the first and last position where a range is found.

8.3.2.3 Multimap Specialized Algorithms

The multimap class provides a specialized function for swapping elements.

8.3.2.3.1 swap

Swaps the first argument for the last argument.
8.3.3 Template Class Set

The template class set is a container that supports unique keys and provides for fast retrieval of the keys themselves.

Set supports bidirectional iterators.

The class set satisfies all of the requirements of a container, a reversible container and an associative container.

A set supports the a_uniq operations but not the a_eq operations.

8.3.3.1 Constructors

The set class includes overloaded constructors for creation of a set object.

explicit set
{const Compare& comp = Compare(),
 const Allocator& = Allocator();
template <class InputIterator> set
{InputIterator first, last,
 const Compare& comp = Compare(),
 const Allocator& = Allocator();

8.3.3.2 Set Specialized Algorithms

The set class specializes the swap function.

8.3.3.2.1 swap

Swaps the first argument with the second argument.
### 8.3.4 Template Class Multiset

The template class multiset is an associative container that supports equivalent keys and retrieval of the keys themselves.

Multiset supports bidirectional iterators.

The multiset satisfies all of the requirements of a container, reversible container and an associative container.

A multiset supports the \texttt{a\_eq} operations but not the \texttt{a\_uniq} operations.

#### 8.3.4.1 Constructors

The multiset class includes overloaded constructors for creation of a multiset object.

```cpp
explicit multiset
    (const Compare& comp = Compare(),
    const Allocator& = Allocator());

template <class InputIterator> multiset
    (InputIterator first, last, const Compare& comp = Compare(),
    const Allocator& = Allocator());
```

#### 8.3.4.2 Multiset Specialized Algorithms

The multiset class provides a specialized swap function.

##### 8.3.4.2.1 swap

Swaps the first argument with the second argument.

```cpp
template <class Key, class Compare, class Allocator>
void swap
    (multiset<Key,Compare,Allocator>& x,
     multiset<Key,Compare,Allocator>& y);
```
8.3.5 Template Class Bitset

The `bitset` header defines a template class and related procedures for representing and manipulating fixed-size sequences of bits.

The template class `bitset` can store a sequence consisting of a fixed number of bits. In the bitset class each bit represents either the value zero (reset) or one (set), there is no negative position. You can toggle a bit to change the value.

When converting between an object of class `bitset` and an integral value, the integral value corresponding to two or more bits is the sum of their bit values.

The `bitset` functions can report three kinds of errors as exceptions.

- An `invalid_argument` exception
- An `out_of_range` error exception
- An `overflow_error` exceptions

See Exception Classes, for more information on exception classes.

8.3.5.1 Constructors

The `bitset` class includes overloaded constructors for creation of a `bitset` object.

```cpp
bitset();
bitset(unsigned long val);
template <class charT, class traits, class Allocator>
explicit bitset
(const basic_string<charT, traits, Allocator>& str,
typeid basic_string
<charT, traits, Allocator>::size_type pos = 0,
typeid basic_string<charT, traits, Allocator>::npos);
```

8.3.5.2 Bitset Members

The `bitset` class provides various member operators.

8.3.5.2.1 `operator &=`
A bitwise "and equal" operator.

```cpp
bitset<N>& operator&=(const bitset<N>& rhs);
```

**Remarks**

Returns the result of the "and equals" operation.

### 8.3.5.2.2 `operator |=`

An "Assignment by bitwise OR" operator.

```cpp
bitset<N>& operator|=(const bitset<N>& rhs);
```

**Remarks**

Assigns the result of the "bitwise OR" of the left and right operands to the left operand.

### 8.3.5.2.3 `operator ^=`

A bitwise "exclusive or equals" operator.

```cpp
bitset<N>& operator^=(const bitset<N>& rhs);
```

**Remarks**

Returns the result of the "exclusive or equals" operation.

### 8.3.5.2.4 `operator <<=`

A bitwise "left shift equals" operator.

```cpp
bitset<N>& operator <<= (size_t pos);
```

**Remarks**

Returns the result of the "left shift equals" operation.

### 8.3.5.2.5 `operator >>=`

A bitwise "right shift equals" operator.

```cpp
bitset<N>& operator >>= (size_t pos);
```

**Remarks**

Returns the result of the "right shift equals" operation.
A bitwise "right shift equals" operator.

```cpp
bitset<N>& operator>>=(size_t pos);
```

**Remarks**

Returns the result of the "right shifts equals" operation.

### 8.3.5.2.6 Set

Sets all the bits or a single bit to a value.

```cpp
bitset<N>& set();
bitset<N>& set(size_t pos, int val = 1);
```

**Remarks**

For the function with no parameters sets all the bits to true. For the overloaded function with just a position argument sets that bit to true. For the function with both a position and a value sets the bit at that position to the value.

Returns the altered bitset.

### 8.3.5.2.7 reset

Sets the bits to false.

```cpp
bitset<N>& reset();
bitset<N>& reset(size_t pos);
```

**Remarks**

The reset function without any arguments sets all the bits to false. The reset function with an argument sets the bit at that position to false.

Returns the modified bitset.

### 8.3.5.2.8 operator ~

Toggles all bits in the bitset.
bitset<N> operator~() const;

**Remarks**
Returns the modified bitset.

### 8.3.5.2.9 flip

Toggles all the bits in the bitset.

```
bitset<N>& flip();
bitset<N>& flip(size_t pos);
```

**Remarks**
Returns the modified bitset.

### 8.3.5.2.10 to_ulong

Gives the value as an unsigned log.

```
unsigned long to_ulong() const;
```

**Remarks**
Returns the unsigned long value that the bitset represents.

### 8.3.5.2.11 to_string

Gives the string as zero and ones that the bitset represents.

```
template <class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const;
```

**Remarks**
Returns a string that the bitset represents.

### 8.3.5.2.12 count


Associative Containers

Tells the number of bits that are true.

```cpp
size_t count() const;
```

**Remarks**

Returns the number of set bits.

**8.3.5.2.13 size**

Tells the size of the bitset as the number of bits.

```cpp
size_t size() const;
```

**Remarks**

Returns the size of the bitset.

**8.3.5.2.14 operator ==**

The equality operator.

```cpp
bool operator==(const bitset<N>& rhs) const;
```

**Remarks**

Returns true if the argument is equal to the right side bitset.

**8.3.5.2.15 operator !=**

The inequality operator.

```cpp
bool operator!=(const bitset<N>& rhs) const;
```

**Remarks**

Returns true if the argument is not equal to the right side bitset.

**8.3.5.2.16 test**
Test if a bit at a position is set.

    bool test(size_t pos) const;

Remarks
Returns true if the bit at the position is true.

8.3.5.2.17 any

Tests if all bits are set to true.

    bool any() const;

Remarks
Returns true if any bits in the bitset are true.

8.3.5.2.18 none

Tests if all bits are set to false.

    bool none() const;

Remarks
Returns true if all bits are false.

8.3.5.2.19 operator <<

Shifts the bitset to the left a number of positions.

    bitset<N> operator<<(size_t pos) const;

Remarks
Returns the modified bitset.

8.3.5.2.20 operator >>
Shifts the bitset to the right a number of positions.

```cpp
bitset<N> operator>>(size_t pos) const;
```

**Remarks**

Returns the modified bitset.

### 8.3.5.3 Bitset Operators

Bitwise operators are included in the bitset class.

#### 8.3.5.3.1 operator &

A bitwise and operator.

```cpp
bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs);
```

**Remarks**

Returns the modified bitset.

#### 8.3.5.3.2 operator |

A bitwise or operator.

```cpp
bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs);
```

**Remarks**

Returns the modified bitset.

#### 8.3.5.3.3 operator ^

A bitwise exclusive or operator.
bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs);

Remarks
Returns the modified bitset.

8.3.5.3.4 operator >>

An extractor operator for a bitset input.

```cpp
template <class charT, class traits, size_t N>
basic_istream<charT, traits>& operator>>(
    basic_istream<charT, traits>& is, bitset<N>& x);
```

Remarks
Returns the bitset.

8.3.5.3.5 operator <<

An inserter operator for a bitset output.

```cpp
template <class charT, class traits, size_t N>
basic_ostream<charT, traits>& operator<<(
    basic_ostream<charT, traits>& os, const bitset<N>& x);
```

Remarks
Returns the bitset.
Chapter 9
Iterators Library

This chapter presents the concept of iterators in detail, defining and illustrating the five iterator categories of input iterators, output iterators, forward iterators, bidirectional iterators and random access iterators.

This chapter describes the components used in C++ programs to perform iterations for container classes, streams and stream buffers.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Requirements
- Header iterator
- Iterator Primitives
- Predefined Iterators
- Stream Iterators
- _EWL_RAW_ITERATORS

9.1 Requirements

Iterators are a generalized pointer that allow the C++ program to work with various containers in a unified manner.

All iterators allow the dereference into a value type.

Since iterators are an abstraction of a pointer all functions that work with regular pointers work equally with regular pointers.

9.1.1 Input Iterators
There are requirements for input iterators, this manual, does not attempt to list them all. Algorithms on input iterators should never attempt to pass through the same iterator more than once.

### 9.1.2 Output Iterators

There are requirements for output iterators, this manual, does not attempt to list them all. An output iterator is assignable.

### 9.1.3 Forward Iterators

Forward iterators meet all the requirements of input and output iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.

### 9.1.4 Bidirectional Iterators

Bidirectional iterators meet the requirements of forward iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.

### 9.1.5 Random Access Iterators

Random access iterators meet the requirements of bidirectional iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.
9.2 Header iterator

The header iterator includes classes, types and functions used to allow the C++ program to work with various containers in a unified manner.

9.3 Iterator Primitives

The library provides several classes and functions to simplify the task of defining iterators.

9.3.1 Iterator Traits

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types for a particular iterator type. Therefore, it is required that if \texttt{iterator} is the type of an iterator, then the types

\begin{verbatim}
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::iterator_category
\end{verbatim}

are defined as the iterator's difference type, value type and iterator category, respectively. In the case of an output iterator, the types

\begin{verbatim}
iterator_traits<Iterator>::difference_type

iterator_traits<Iterator>::value_type
\end{verbatim}

defined as void.

The \texttt{template} \texttt{iterator_traits<Iterator>} is specialized for pointers and for pointers to const

9.3.2 Basic Iterator

The iterator template may be used as a base class for new iterators.
9.3.3 Standard Iterator Tags

The standard library includes category tag classes which are used as compile time tags for algorithm selection. These tags are used to determine the best iterator argument at compile time. These tags are:

- `input_iterator_tag`
- `output_iterator_tag`
- `forward_iterator_tag`
- `bidirectional_iterator_tag`
- `random_access_iterator_tag`

9.3.4 Iterator Operations

Since only random access iterators provide plus and minus operators, the library provides two template functions for this functionality.

9.3.4.1 advance

Increments or decrements iterators.

```cpp
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
```

9.3.4.2 distance

Provides a means to determine the number of increments or decrements necessary to get from the beginning to the end.

```cpp
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type distance
(InputIterator first, InputIterator last);
```

Remarks

The distance from last must be reachable from first.

The number of increments from first to last.
9.4 Predefined Iterators

The standard provides for two basic predefined iterators.

- Reverse iterators
- Insert Iterators

9.4.1 Reverse iterators

Both bidirectional and random access iterators have corresponding reverse iterator adaptors that they iterate through.

9.4.1.1 Template Class Reverse_iterator

A reverse_iterator must meet the requirements of a bidirectional iterator.

9.4.1.2 Reverse_iterator Requirements

Additional requirements may be necessary if random access operators are referenced in a way that requires instantiation.

9.4.1.3 Constructors

Creates an instance of a reverse_iterator object.

```cpp
explicit reverse_iterator(Iterator x);
template <class U> reverse_iterator
    (const reverse_iterator<U> &u);
```

9.4.1.4 base
The base operator is used for conversion.

```
Iterator base() const; // explicit
```

**Remarks**

The current iterator is returned.

### 9.4.1.5 Reverse_iterator operators

The common operators are provided for `reverse_iterator`.

**Operator**

```
reference operator*() const;
```

A reference iterator is returned.

A pointer to the dereferenced iterator is returned.

**Operator->**

```
pointer operator ->() const;
```

**Operator++**

```
reverse_iterator& operator++();
reverse_iterator operator++(int);
```

The this pointer is returned.

**Operator--**

```
reverse_iterator& operator--();
reverse_iterator operator--(int);
```

The this pointer is returned.

**Operator+**

```
reverse_iterator operator+( typename reverse_iterator<Iterator>::difference_type n) const;
```

The reverse_iterator representing the result of the operation is returned.
Operator+=

```cpp
reverse_iterator& operator+=
(typename reverse_iterator<Iterator>::difference_type n);
```

The reverse_iterator representing the result of the operation is returned.

Operator-

```cpp
iterator operator-
(typename reverse_iterator<Iterator>::difference_type n) const;
```

The reverse_iterator representing the result of the operation is returned.

Operator-=

```cpp
reverse_iterator& operator-=
(typename reverse_iterator<Iterator>::difference_type n);
```

The reverse_iterator representing the result of the operation is returned.

Operator[]

```cpp
reference operator[]
(typename reverse_iterator<Iterator>::difference_type n) const;
```

An element access reference is returned.

Operator==

```cpp
template <class Iterator>
bool operator==
(const reverse_iterator<Iterator>& x,
 const reverse_iterator<Iterator>& y);
```

A bool true value is returned if the iterators are equal.

Operator<

```cpp
template <class Iterator>
bool operator<
(const reverse_iterator<Iterator>& x,
 const reverse_iterator<Iterator>& y);
```

A bool true value is returned if the first iterator is less than the second.

Operator!=

```cpp
template <class Iterator>
bool operator!=
(const reverse_iterator<Iterator>& x,
 const reverse_iterator<Iterator>& y);
```
template <class Iterator> bool operator!=(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);

A bool true value is returned if the first iterator is not equal to the second.

Operator>

template <class Iterator> bool operator>(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);

A bool true value is returned if the first iterator is greater than the second.

Operator>=

template <class Iterator> bool operator>=(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);

The reverse_iterator representing the result of the operation is returned.

Operator<=

template <class Iterator> bool operator<=(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);

The reverse_iterator representing the result of the operation is returned.

Operator-

template <class Iterator> typename reverse_iterator<Iterator>::difference_type operator-(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);

The reverse_iterator representing the result of the operation is returned.

Operator+

template <class Iterator> reverse_iterator<Iterator> operator+(typename reverse_iterator<Iterator>::difference_type n, const reverse_iterator<Iterator>& x);

The reverse_iterator representing the result of the operation is returned.
9.4.2 Insert Iterators

Insert iterators, are provided to make it possible to deal with insertion in the same way as writing into an array.

9.4.2.1 Class back_insert_iterator

A back_insert_iterator inserts at the back.

9.4.2.2 Constructors

Constructs a back_insert_iterator object.

    explicit back_insert_iterator(Container& x);

9.4.2.2.1 operator =

An operator is provided for copying a const_reference value.

    back_insert_iterator<Container>& operator= (typename Container::const_reference value);

Remarks

A reference to the copied back_insert_iterator is returned.

9.4.2.3 Back_insert_iterator Operators

Several standard operators are provided for Back_insert_iterator.
9.4.2.3.1 Operator *

\[
\text{back_insert_iterator<Container>& operator*();}
\]

The dereferenced iterator is returned.

9.4.2.3.2 Operator ++

\[
\text{back_insert_iterator<Container>& operator++();}
\]
\[
\text{back_insert_iterator<Container> operator++(int);}
\]

The incremented iterator is returned.

9.4.2.4 back_inserter

Provides a means to get the back iterator.

\[
\text{template <class Container> back_insert_iterator<Container>
back_inserter}
\]
\[
\text{(Container& x);}
\]

Remarks

The back_inserter is returned.

9.4.3 Template Class Front_insert_iterator

A front_insert_iterator inserts at the front.

9.4.3.1 Constructors

Creates a front_insert_iterator object.

\[
\text{explicit front_insert_iterator(Container& x);}\]
Operator=
Assigns a value to a `front_insert_iterator` object.

```cpp
front_insert_iterator<Container>& operator=
(typename Container::const_reference value);
```

Remarks
A `front_insert_iterator` copy of the `const_reference` value is returned.

### 9.4.3.2 Front_insert_iterator operators

Several common operators are provided for the `front_insert_iterator` class.

**Operator***

```cpp
front_insert_iterator<Container>& operator*();
```

A this pointer is returned.

**Operator++**

```cpp
front_insert_iterator<Container>& operator++();
```

```cpp
front_insert_iterator<Container> operator++(int);
```

Remarks
A post or pre increment operator.
The this pointer is returned.

### 9.4.3.3 front_inserter

Provides a means to get the front iterator.
template <class Container>
    front_insert_iterator<Container>
    front_inserter(Container& x);

Remarks
The front_insert_iterator is returned.

9.4.4 Template Class Insert_iterator
A bidirectional insertion iterator.

9.4.4.1 Constructors
Creates an instance of an insert_iterator object.

    insert_iterator
    (Container& x, typename Container::iterator i);

operator=
An operator for assignment of a const_reference value.

    insert_iterator<Container>& operator=
    (typename Container::const_reference value);

Remarks
Returns a copy of the insert_iterator.

9.4.4.2 Insert_iterator Operators
Various operators are provided for an insert_iterator.

Operator*

    insert_iterator<Container>& operator*();

The dereferenced iterator is returned.
Operator++

```cpp
insert_iterator<Container>& operator++();
insert_iterator<Container>& operator++(int);
```

The this pointer is returned.

### 9.4.4.3 inserter

Provides a means to get the iterator.

```cpp
template <class Container, class Inserter>
insert_iterator<Container> inserter (Container& x, Inserter i);
```

**Remarks**

The inserter iterator is returned.

### 9.5 Stream Iterators

Input and output iterators are provided to make it possible for algorithmic templates to work directly with input and output streams.

#### 9.5.1 Template Class Istream_iterator

An istream_iterator reads (using `operator>>`) successive elements from the input stream. It reads after it is constructed, and every time the increment operator is used.

If an end of stream is reached the iterator returns false.

Since istream iterators are not assignable istream iterators can only be used for one pass algorithms.
9.5.1.1 Constructors

Creates and object of an `istream_iterator` object.

```cpp
istream_iterator();
istream_iterator(istream_type& s);
istream_iterator(const istream_iterator<T, charT,traits,Distance>& x);
```

The parameterless iterator is the only legal constructor for an `end` condition.

9.5.1.2 Destructor

Removes an instance of an `istream_iterator`.

```cpp
~istream_iterator();
```

9.5.1.3 Istream_iterator Operations

Various operators are provided for an `istream_iterator`.

**Operator**

```cpp
const T& operator*() const;
```

A dereferenced iterator is returned.

**Operator->**

```cpp
const T* operator->() const;
```

The address of a dereferenced iterator is returned.

**Operator++**

```cpp
istream_iterator <T,charT,traits,Distance>& operator++();
istream_iterator <T,charT,traits,Distance>& operator++(int);
```

The this pointer is returned.
Operator==

```
template <class T, class charT, class traits, class Distance> bool operator==
(const istream_iterator<T,charT,traits,Distance> & x, const istream_iterator<T,charT,traits,Distance> & y);
```

A bool true value is returned if the arguments are the same.

### 9.5.2 Template Class Ostring_iterator

The ostream_iterator writes (using `operator<<<`) successive elements onto the output stream.

#### 9.5.2.1 Constructors

Creates and instance of an ostream_iterator object.

```
ostream_iterator(ostream_type& s);
ostream_iterator(ostream_type& s, const charT* delimiter);
ostream_iterator(const ostream_iterator& x);
```

**Operator=**

```
ostream_iterator& operator=(const T& value);
```

Returns a value to an ostream iterator.

#### 9.5.2.2 Destructor

Removes an instance of an ostream_iterator object.

```
~ostream_iterator();
```

#### 9.5.2.3 Ostring_iterator Operators

Various operators are provided in the ostream_iterator class.
Stream Iterators

Operator*

```cpp
ostream_iterator& operator*();
```

The dereference iterator is returned.

Operator++

```cpp
ostream_iterator& operator++();
ostream_iterator& operator++(int);
```

The this pointer is returned.

### 9.5.3 Template Class istreambuf_iterator

The `istreambuf_iterator` reads successive characters from the `istreambuf` object for which it was constructed.

An `istreambuf_iterator` can only be used for a one pass algorithm.

#### 9.5.3.1 Constructors

An overloaded constructor is provided for creation of an `istreambuf_iterator` object.

```cpp
istreambuf_iterator() throw();
istreambuf_iterator(basic_istream<CharT,traits>& s) throw();
istreambuf_iterator(basic_streambuf<CharT,traits>* s) throw();
istreambuf_iterator(const proxy& p) throw();
```

#### 9.5.3.2 Istreambuf_iterator Operators

Various operators are provided for the `istreambuf_iterator` class.

Operator*

```cpp
CharT operator*() const
```
A dereferenced character type is returned.

**Operator**

```cpp
istreambuf_iterator<charT,traits>&
istreambuf_iterator<charT,traits>::operator++();
```

The this pointer is returned.

**Operator**

```cpp
template <class charT, class traits>
bool operator==
(const istreambuf_iterator<charT,traits>& a,
const istreambuf_iterator<charT,traits>& b);
```

True is returned if the arguments are equal.

**Operator**

```cpp
template <class charT, class traits>
bool operator!=
(const istreambuf_iterator<charT,traits>& a,
const istreambuf_iterator<charT,traits>& b);
```

True is returned if the arguments are not equal.

## 9.5.3.3  equal

An equality comparison.

```cpp
bool equal(istreambuf_iterator<charT,traits>& b);
```

**Remarks**

True is returned if the arguments are equal.

## 9.5.4  Template Class Ostreaembuf_iterator

The `ostreaembuf_iterator` writes successive characters to the ostreambuf object for which it was constructed.
9.5.4.1 Constructors

The constructor is overloaded for creation of an `ostreambuf_iterator` object.

```cpp
ostreambuf_iterator(ostream_type& s) throw();
ostreambuf_iterator(streambuf_type* s) throw();
```

**Operator=**

```cpp
ostreambuf_iterator<charT,traits>& operator=(charT c);
```

The result of the assignment is returned.

9.5.4.2 Ostrembuf_iterator Operators

**Operator***

```cpp
ostreambuf_iterator<charT,traits>& operator*();
```

The dereferenced `ostreambuf_iterator` is returned.

**Operator++**

```cpp
ostreambuf_iterator<charT,traits>& operator++();
ostreambuf_iterator<charT,traits>& operator++(int);
```

The this pointer is returned.

9.5.4.3 failed

Reports a failure in writing.

```cpp
bool failed() const throw();
```

**Remarks**

The bool false value is returned if a write failure occurs.
9.6 _EWL_RAW_ITERATORS

If _EWL_RAW_ITERATORS is defined, vector and string will use pointers for their iterators, otherwise they will use classes.

The difference can effect argument dependent (Koenig) lookup in some cases. For example:

```cpp
#include <vector>
#include <algorithm>
int main()
{
    std::vector<int> v1(10), v2(10);
    copy(v1.begin(), v1.end(), v2.begin());
}
```

This compiles if the iterators are classes (_EWL_RAW_ITERATORS undefined). But if the iterators are simply pointers, a compile time error results:

```
Error : undefined identifier 'copy'
```

To fix this code so that it works with either setting, add a std qualifier to copy:

```cpp
std::copy(v1.begin(), v1.end(), v2.begin());
```

The default configuration is for _EWL_RAW_ITERATORS to be undefined. There is no code size or run time overhead for this configuration (with inlining turned on). If you use _EWL_DEBUG (a configuration that does extensive run time checking when using the STL), then behavior is consistent with a _EWL_RAW_ITERATORS undefined setting, since the use of _EWL_DEBUG also forces vector and string iterators to be classes. Therefore the behavior of your application is less likely to change when switching between debug and release builds.

**NOTE**

Recompile EWL C++ when switching this flag.
Chapter 10
Algorithms Library

This chapter discusses the algorithms library. These algorithms cover sequences, sorting, and numerics.

The standard provides for various algorithms that a C++ program may use to perform algorithmic operations on containers and other sequences.

This chapter uses the ISO (International Organization for Standardization) C++ Standard as a guide.

10.1 Header algorithm

The header algorithm provides classes, types and functions for use with the standard C++ libraries.

The standard algorithms can work with program defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

The names of the parameters used in this chapter reflect their usage.

A predicate parameter is used for a function object that returns a value testable as true. The binary predicate parameter takes two arguments.

10.1.1 Non-modifying Sequence Operations

Various algorithms are provided which do not modify the original object.

10.1.1.1 for_each
The function `for_each` is used to perform an operation for each element.

```cpp
template<class InputIterator, class Function>
Function for_each
(InputIterator first, InputIterator last, Function f);
```

**Remarks**
The function `f` is returned.

### 10.1.1.2 find

The function `find` searches for the first element that contains the value passed.

```cpp
template<class InputIterator, class T>
InputIterator find
(InputIterator first, InputIterator last, const T& value);
```

**Remarks**
Returns the type passed.

### 10.1.1.3 find_if

The function `find_if` searches for the first element that matches the criteria passed by the predicate.

```cpp
template<class InputIterator, class Predicate>
InputIterator find_if
(InputIterator first, InputIterator last, Predicate pred);
```

**Remarks**
Returns the iterator of the matched value.

### 10.1.1.4 find_end

The function `find_end` searches for the last occurrence of a value.
template<class ForwardIterator1, 
class ForwardIterator2>
ForwardIterator1 find_end  
(ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2);  
template<class ForwardIterator1, 
class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_end 
(ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2, 
BinaryPredicate pred);

Remarks
Returns the iterator to the last value or the last1 argument if none is found.

10.1.1.5 find_first_of

The function find_first_of searches for the first occurrence of a value.

template<class ForwardIterator1, 
class ForwardIterator2>
ForwardIterator1 find_first_of 
(ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2);  
template<class ForwardIterator1, 
class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_first_of 
(ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);

Remarks
Returns the iterator to the first value or the last1 argument if none is found.

10.1.1.6 adjacent_find

The function adjacent_find is used to search for two adjacent elements that are equal or equal according to the predicate argument.

template<class ForwardIterator>  
ForwardIterator adjacent_find 
(ForwardIterator first, ForwardIterator last);  
template<class ForwardIterator, class BinaryPredicate> 
ForwardIterator adjacent_find 
(ForwardIterator first, ForwardIterator last, 
BinaryPredicate pred);
Remarks
Returns the iterator to the first occurrence found or to \texttt{last} if no occurrence is found.

10.1.1.7 \hspace{1em} \texttt{count}

The function \texttt{count} is used to find the number of elements.

\begin{verbatim}
template <class InputIterator, class T>
    typename iterator_traits<InputIterator>::difference_type count
    (InputIterator first, InputIterator last, const T& value);
\end{verbatim}

Remarks
Returns the number of elements (iterators) as an
\texttt{iterator_traits<InputIterator>::difference_type}

10.1.1.8 \hspace{1em} \texttt{count_if}

The function \texttt{count_if} is used to find the number of elements that match the criteria.

\begin{verbatim}
template <class InputIterator, class Predicate>
    typename iterator_traits<InputIterator>::difference_type count_if
    (InputIterator first, InputIterator last, Predicate pred);
\end{verbatim}

Remarks
Returns the number of elements (iterators) as an
\texttt{iterator_traits<InputIterator>::difference_type}

10.1.1.9 \hspace{1em} \texttt{mismatch}

The function \texttt{mismatch} is used to find sequences that are not the same or differ according to the predicate criteria.

\begin{verbatim}
template<class InputIterator1, class InputIterator2>
    pair<InputIterator1, InputIterator2> mismatch
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2);
\end{verbatim}
template<class InputIterator1, class InputIterator2, class BinaryPredicate> 
pair<InputIterator1, InputIterator2> mismatch 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, BinaryPredicate pred);

**Remarks**

Returns a pair<iterator> that represent the beginning element and the range. If no 
mismatch is found the end and the corresponding range element is returned.

### 10.1.1.10 equal

The function **equal** is used to determine if two ranges are equal.

```
template<class InputIterator1, class InputIterator2> 
bool equal 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2);
template<class InputIterator1, 
class InputIterator2, class BinaryPredicate> 
bool equal 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, BinaryPredicate pred);
```

**Remarks**

A bool true is returned if the values are equal or meet the criteria of the predicate.

### 10.1.1.11 search

The function **search** is used to search for the first occurrence of a sub-range that meets the 
criteria.

```
template<class ForwardIterator1, class ForwardIterator2> 
ForwardIterator1 search 
(ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, 
class ForwardIterator2, class BinaryPredicate> 
ForwardIterator1 search 
(ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2, 
BinaryPredicate pred);
```

**Remarks**

An iterator to the first occurrence is returned or last1 is returned if no criteria is met.
10.1.1.12  **search_n**

The function `search_n` is used to search for a number of consecutive elements with the same properties.

```cpp
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n
(ForwardIterator first, ForwardIterator last,
 Size count, const T& value);

template<class ForwardIterator,
class Size, class T, class BinaryPredicate>
ForwardIterator search_n
(ForwardIterator first, ForwardIterator last, Size count,
 const T& value, BinaryPredicate pred);
```

**Remarks**

An iterator to the first occurrence is returned or `last1` is returned if no criteria is met.

10.1.2  **Mutating Sequence Operators**

Various algorithms are provided that are used to modify the original object.

10.1.2.1  `copy`

The function `copy` is used to copy a range.

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first,
 InputIterator last,OutputIterator result);
```

**Remarks**

The position of the last copied element is returned.

10.1.2.2  `copy_backward`

The function `copy_backwards` is used to copy a range starting with the last element.
template<class BidirectionalIterator1, 
       class BidirectionalIterator2>
BidirectionalIterator2 copy_backward
    (BidirectionalIterator1 first, BidirectionalIterator1 last, 
     BidirectionalIterator2 result);

Remarks
The position of the last copied element is returned.

10.1.2.3 swap

The function swap is used to exchange values from two locations.

    template<class T> void swap(T & a, T & b);

Remarks
There is no return.

10.1.2.4 swap_ranges

The function swap_ranges is used swap elements of two ranges.

    template<class ForwardIterator1, class ForwardIterator2>
    ForwardIterator2 swap_ranges
        (ForwardIterator1 first1, ForwardIterator1 last1, 
         ForwardIterator2 first2);

Remarks
The position of the last swapped element is returned.

10.1.2.5 iter_swap

The function iter_swap is used to exchange two values pointed to by iterators.

    template<class ForwardIterator1, class ForwardIterator2>
    void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
Remarks
There is no return.

10.1.2.6 transform

The function transform is used to modify and copy elements of two ranges.

```cpp
template<class InputIterator, 
    class OutputIterator, class UnaryOperation>
OutputIterator transform 
    (InputIterator first, InputIterator last, 
    OutputIterator result, UnaryOperation op);

template<class InputIterator1, 
    class InputIterator2, class OutputIterator, 
    class BinaryOperation>
OutputIterator transform 
    (InputIterator1 first1, InputIterator1 last1, 
    InputIterator2 first2, OutputIterator result, 
    BinaryOperation binary_op);
```

Remarks
The position of the last transformed element is returned.

10.1.2.7 replace

The function replace is used to replace an element with another element of different value.

```cpp
template<class ForwardIterator, class T>
void replace 
    (ForwardIterator first, ForwardIterator last, 
    const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
void replace_if 
    (ForwardIterator first, ForwardIterator last, 
    Predicate pred, const T& new_value);
```

Remarks
There is no return.

10.1.2.8 replace_copy
The function `replace_copy` is used to replace specific elements while copying an entire range.

```cpp
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy
(InputIterator first, InputIterator last,
OutputIterator result,
const T& old_value, const T& new_value);
```

**Remarks**
The position of the last copied element is returned.

### 10.1.2.9 replace_copy_if

The function `replace_copy_if` is used to replace specific elements that match certain criteria while copying the entire range.

```cpp
template<class Iterator,
class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if
(Iterator first, Iterator last,
OutputIterator result, Predicate pred, const T& new_value);
```

**Remarks**
The position of the last copied element is returned.

### 10.1.2.10 fill

The function `fill` is used to fill a range with values.

```cpp
template<class ForwardIterator, class T> 
void fill
(ForwardIterator first, ForwardIterator last, const T& value);
```

**Remarks**
There is no return value.
10.1.2.11 fill_n

The function `fill_n` is used to fill a number of elements with a specified value.

```cpp
template<class OutputIterator, class Size, class T>
void fill_n
(OutputIterator first, Size n, const T& value);
```

**Remarks**

There is no return value.

10.1.2.12 generate

The function `generate` is used to replace elements with the result of an operation.

```cpp
template<class ForwardIterator, class Generator>
void generate
(ForwardIterator first, ForwardIterator last, Generator gen);
```

**Remarks**

There is no return value.

10.1.2.13 generate_n

The function `generate_n` is used to replace a number of elements with the result of an operation.

```cpp
template<class OutputIterator, class Size, class Generator>
void generate_n
(OutputIterator first, Size n, Generator gen);
```

**Remarks**

There is no return value.

10.1.2.14 remove
The function \texttt{remove} is used to remove elements with a specified value.

\begin{verbatim}
template<class ForwardIterator, class T>
ForwardIterator remove
  (ForwardIterator first, ForwardIterator last,const T& value);
\end{verbatim}

\textbf{Remarks}

The end of the resulting range is returned.

\subsection*{10.1.2.15 \texttt{remove\_if}}

The function \texttt{remove\_if} is used to remove elements using a specified criteria.

\begin{verbatim}
template<class ForwardIterator, class Predicate>
ForwardIterator remove\_if
  (ForwardIterator first, ForwardIterator last,Predicate pred);
\end{verbatim}

\textbf{Remarks}

The end of the resulting range is returned.

\subsection*{10.1.2.16 \texttt{remove\_copy}}

The function \texttt{remove\_copy} is used remove elements that do not match a value during a copy.

\begin{verbatim}
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove\_copy
  (InputIterator first, InputIterator last,
   OutputIterator result, const T& value);
\end{verbatim}

\textbf{Remarks}

The end of the resulting range is returned.

\subsection*{10.1.2.17 \texttt{remove\_copy\_if}}

The function \texttt{remove\_copy\_if} is used to remove elements that do not match a criteria while doing a copy.
template<class InputIterator,  
class OutputIterator, class Predicate>  
OutputIterator remove_copy_if  
(InputIterator first, InputIterator last,  
OutputIterator result, Predicate pred);

Remarks
The end of the resulting range is returned.

10.1.2.18 unique

The function unique is used to remove all adjacent duplicates.

template<class ForwardIterator>  
ForwardIterator unique  
(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>  
ForwardIterator unique (ForwardIterator first,  
ForwardIterator last, BinaryPredicate pred);

Remarks
The end of the resulting range is returned.

10.1.2.19 unique_copy

The function unique_copy is used to remove adjacent duplicates while copying.

template<class InputIterator, class OutputIterator>  
OutputIterator unique_copy  
(InputIterator first, InputIterator last,  
OutputIterator result);

template<class InputIterator,  
class OutputIterator, class BinaryPredicate>  
OutputIterator unique_copy  
(InputIterator first, InputIterator last,  
OutputIterator result, BinaryPredicate pred);

Remarks
The end of the resulting range is returned.
10.1.2.20 reverse

The function `reverse` is used to reverse a sequence.

```
template<class BidirectionalIterator>
void reverse
(BidirectionalIterator first,BidirectionalIterator last);
```

Remarks

No value is returned.

10.1.2.21 reverse_copy

The function `reverse_copy` is used to copy the elements while reversing their order.

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy
(BidirectionalIterator first,BidirectionalIterator last,
 OutputIterator result);
```

Remarks

The position of the last copied element is returned.

10.1.2.22 rotate

The function `rotate` is used to rotate the elements within a sequence.

```
template<class ForwardIterator>
void rotate
(ForwardIterator first, ForwardIterator middle,
 ForwardIterator last);
```

Remarks

There is no return value.

10.1.2.23 rotate_copy
The function **rotate_copy** is used to copy a sequence with a rotated order.

```cpp
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy
(ForwardIterator first, ForwardIterator middle,
 ForwardIterator last, OutputIterator result);
```

**Remarks**

The position of the last copied element is returned.

### 10.1.2.24 random_shuffle

The function **random_shuffle** is used to exchange the order of the elements in a random fashion.

```cpp
template<class RandomAccessIterator>
void random_shuffle
(RandomAccessIterator first,RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator,
 class RandomNumberGenerator>
void random_shuffle
(RandomAccessIterator first, RandomAccessIterator last,
 RandomNumberGenerator& rand);
```

**Remarks**

No value is returned.

### 10.1.2.25 partition

The function **partition** is used to change the order of the elements so that the elements that meet the criteria are first in order.

```cpp
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition
(BidirectionalIterator first,
 BidirectionalIterator last, Predicate pred);
```

**Remarks**

Returns an iterator to the first position where the predicate argument is false.
10.1.2.26 stable_partition

The function \texttt{stable\_partition} is used to change the order of the elements so that the elements meet the criteria are first in order. The relative original order is preserved.

\begin{verbatim}
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition
(BidirectionalIterator first,
BidirectionalIterator last, Predicate pred);
\end{verbatim}

\textbf{Remarks}

Returns an iterator to the first position where the predicate argument is false.

10.1.3 Sorting And Related Operations

All of the sorting functions have two versions: one that takes a function object for comparison and one that uses the less than operator.

10.1.3.1 sort

The function \texttt{sort} is used sorts the range according to the criteria.

\begin{verbatim}
template<class RandomAccessIterator>
void sort
(RandomAccessIterator first,RandomAccessIterator last);
template<class RandomAccessIterator,
class Compare>
void sort(RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
\end{verbatim}

\textbf{Remarks}

There is no return value.

10.1.3.2 stable_sort

The function \texttt{stable\_sort} is used to sort the range but preserves the original order for equal elements.
template<class RandomAccessIterator>
void stable_sort
(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort
(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

**Remarks**

There is no return value.

### 10.1.3.3 partial_sort

The function partial_sort is used to sort a sub-range leaving the rest unsorted.

template<class RandomAccessIterator>
void partial_sort
(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort
(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp);

**Remarks**

There is no return value.

### 10.1.3.4 partial_sort_copy

The function partial_sort_copy is used to copy a partially sorted sequence.

template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy
(InputIterator first, InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy
(InputIterator first, InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last, Compare comp);

**Remarks**

The position at the end of the copied elements is returned.
10.1.3.5   nth_element

The function \texttt{nth_element} is used to sort based upon a specified position.

\begin{verbatim}
template<class RandomAccessIterator>
void nth_element
(RandomAccessIterator first RandomAccessIterator nth,
 RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void nth_element
(RandomAccessIterator first,RandomAccessIterator nth,
 RandomAccessIterator last, Compare comp);
\end{verbatim}

\textbf{Remarks}

There is no value returned.

10.1.3.6   lower_bound

The function \texttt{lower_bound} is used to find the first position that an element may be inserted without changing the order.

\begin{verbatim}
template<class ForwardIterator, class T>
ForwardIterator lower_bound
(ForwardIterator first, ForwardIterator last, const T&
 value);
template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound
(ForwardIterator first, ForwardIterator last,
 const T& value, Compare comp);
\end{verbatim}

\textbf{Remarks}

The position where the element can be inserted is returned.

10.1.3.7   upper_bound

The function \texttt{upper_bound} is used to find the last position that an element may be inserted without changing the order.

\begin{verbatim}
template<class ForwardIterator, class T>
ForwardIterator upper_bound
(ForwardIterator first, ForwardIterator last, const T&
 value);
\end{verbatim}
value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound
(ForwardIterator first, ForwardIterator last, 
const T& value, Compare comp);

Remarks
The position where the element can be inserted is returned.

### 10.1.3.8 equal_range

The function `equal_range` is used to find the range as a pair where an element can be inserted without altering the order.

```cpp
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range
(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range
(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);
```

Remarks
The range as a pair<> where the element can be inserted is returned.

### 10.1.3.9 binary_search

The function `binary_search` is used to see if a value is present in a range or that a value meets a criteria within that range.

```cpp
template<class ForwardIterator, class T>
bool binary_search
(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search
(ForwardIterator first, ForwardIterator last, 
const T& value, Compare comp);
```

Remarks
The bool value true is met if any element meets the criteria.
10.1.3.10  merge

The function **merge** is used to combine two sorted ranges.

```cpp
template<class InputIterator1, 
  class InputIterator2, class OutputIterator> 
OutputIterator merge 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, InputIterator2 last2, 
OutputIterator result);
template<class InputIterator1, class InputIterator2, 
class OutputIterator, class Compare> 
OutputIterator merge 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, InputIterator2 last2, 
OutputIterator result, Compare comp);
```

**Return**

The position of the first element not overwritten is returned.

10.1.3.11  inplace_merge

The function **inplace_merge** is used to merge consecutive sequences to the first for a concatenation.

```cpp
template<class BidirectionalIterator> 
void inplace_merge 
(BidirectionalIterator first,BidirectionalIterator middle, 
BidirectionalIterator last); 
template<class BidirectionalIterator, class Compare> 
void inplace_merge 
(BidirectionalIterator first,BidirectionalIterator middle, 
BidirectionalIterator last, Compare comp);
```

**Remarks**

There is no value returned.

10.1.3.12  includes

The function **includes** is used to determine if every element meets a specified criteria.

```cpp
template<class InputIterator1, class InputIterator2> 
bool includes 
(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1,
    class InputIterator2, class Compare>
    bool includes
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     Compare comp);

Remarks
The bool value true is returned if all values match or false if one or more does not meet the criteria.

10.1.3.13 set_union

The function `set_union` is used to process the sorted union of two ranges.

template<class InputIterator1,
    class InputIterator2, class OutputIterator>
    OutputIterator set_union
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result);
template<class InputIterator1, class InputIterator2,
    class OutputIterator, class Compare>
    OutputIterator set_union
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result, Compare comp);

Remarks
The end of the constructed range is returned.

10.1.3.14 set_intersection

The function `set_intersection` is used to process the intersection of two ranges.

template<class InputIterator1,
    class InputIterator2, class OutputIterator>
    OutputIterator set_intersection
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result);
InputIterator2 first2, InputIterator2 last2, 
OutputIterator result, Compare comp);

Remarks
The end of the constructed range is returned.

10.1.3.15 set_difference

The function `set_difference` is used to process all of the elements of one range that are not part of another range.

```
template<class InputIterator1, 
class InputIterator2, class OutputIterator> 
OutputIterator set_difference 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, InputIterator2 last2, 
OutputIterator result); 
template<class InputIterator1, 
class InputIterator2, 
class OutputIterator, class Compare> 
OutputIterator set_difference 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, InputIterator2 last2, 
OutputIterator result, Compare comp);
```

Remarks
The end of the constructed range is returned.

10.1.3.16 set_symmetric_difference

The function `set_symmetric_difference` is used to process all of the elements that are in only one of two ranges.

```
template<class InputIterator1, 
class InputIterator2, class OutputIterator> 
OutputIterator set_symmetric_difference 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, InputIterator2 last2, 
OutputIterator result); 
template<class InputIterator1, 
class InputIterator2, 
class OutputIterator, class Compare> 
OutputIterator set_symmetric_difference 
(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, InputIterator2 last2, 
OutputIterator result, Compare comp);
```
Remarks
The end of the constructed range is returned.

10.1.3.17 push_heap

The function push_heap is used to add an element to a heap.

```cpp
template<class RandomAccessIterator>
void push_heap
(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap
(RandomAccessIterator first,
RandomAccessIterator last,Compare comp);
```

Remarks
There is no value returned.

10.1.3.18 pop_heap

The function pop_heap is used to remove an element from a heap.

```cpp
template<class RandomAccessIterator>
void pop_heap
(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap
(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

Remarks
There is no value returned.

10.1.3.19 make_heap

The function make_heap is used to convert a range into a heap.

```cpp
template<class RandomAccessIterator>
void make_heap
(RandomAccessIterator first, RandomAccessIterator last);
```
template<class RandomAccessIterator, class Compare>
void make_heap(
    RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

Remarks
There is no value returned.

10.1.3.20 sort_heap

The function sort_heap is used to sort a heap.

template<class RandomAccessIterator>
void sort_heap
    (RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void sort_heap
    (RandomAccessIterator first, RandomAccessIterator last,
     Compare comp);

Remarks
Note that this result is not stable
There is no value returned.

10.1.3.21 min

The function min is used to determine the lesser of two objects by value or based upon a comparison.

template<class T>
const T& min (const T& a, const T& b);
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);

Remarks
The lesser of the two objects is returned.

10.1.3.22 max
The function \texttt{max} is used to determine the greater of two objects by value or based upon a comparison.

\begin{verbatim}
    template<class T>
    const T& max (const T& a, const T& b);
    template<class T, class Compare>
    const T& max(const T& a, const T& b, Compare comp);
\end{verbatim}

\textbf{Remarks}

The greater of the two objects is returned.

\textbf{10.1.3.23 \ min\_element}

The function \texttt{min\_element} is used to determine the lesser element within a range based upon a value or a comparison.

\begin{verbatim}
    template<class ForwardIterator>
    ForwardIterator min_element
    (ForwardIterator first, ForwardIterator last);
    template<class ForwardIterator, class Compare>
    ForwardIterator min_element
    (ForwardIterator first, ForwardIterator last,
    Compare comp);
\end{verbatim}

\textbf{Remarks}

The position of the element is returned.

\textbf{10.1.3.24 \ max\_element}

The function \texttt{max\_element} is used to determine the greater element within a range based upon a value or a comparison.

\begin{verbatim}
    template<class ForwardIterator>
    ForwardIterator max_element
    (ForwardIterator first, ForwardIterator last);
    template<class ForwardIterator, class Compare>
    ForwardIterator max_element
    (ForwardIterator first, ForwardIterator last,
    Compare comp);
\end{verbatim}

\textbf{Remarks}

The position of the element is returned.
10.1.3.25 lexicographical_compare

The function lexicographical_compare is used to determine if a range is lexicographically less than another.

```cpp
#include <algorithm>

template<class InputIterator1, class InputIterator2>
bool lexicographical_compare
(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare
(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 Compare comp);
```

Remarks

Returns true if the first argument is less than the second and false for all other conditions.

10.1.3.26 next_permutation

The function next_permutation is used to sort in an ascending order based upon lexicographical criteria.

```cpp
#include <algorithm>

template<class BidirectionalIterator>
bool next_permutation
(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation
(BidirectionalIterator first,
 BidirectionalIterator last, Compare comp);
```

Remarks

Returns true if all elements have been sorted.

10.1.3.27 prev_permutation

The function prev_permutation is used to sort in an descending order based upon lexicographical criteria.
template<class BidirectionalIterator>
bool prev_permutation
  (BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation
  (BidirectionalIterator first,
   BidirectionalIterator last, Compare comp);

Remarks

Returns true if all elements have been sorted.

10.1.4 C library algorithms

The C++ header <cstdlib> provides two variations from the standard C header stdlib.h for searching and sorting.

10.1.4.1 bsearch

The function signature of bsearch

bsearch(const void *, const void *, size_t,
size_t,   int (*)(const void *, const void *)));

is replaced by

extern "C" void *bsearch
  (const void * key, const void * base,
size_t nmem, size_t size,
int (* compar)(const void *, const void *)));

and

extern "C++" void *bsearch
  (const void * key, const void * base,
size_t nmem, size_t size,
int (* compar)(const void *, const void *)));

10.1.4.2 qsort

The function signature of qsort
qsort(void *, size_t, size_t,
    int (*)(const void *, const void *));

is replaced by

extern "C" void qsort
    void* base, size_t nmemb, size_t size,
    int (*compar)(const void*, const void*));

and

extern "C++" void qsort
    (void* base, size_t nmemb, size_t size,
    int (*compar)(const void*, const void*));
Chapter 11
Numerics Library

This chapter is a reference guide to the ANSI/ISO standard Numeric classes which are used to perform the semi-numerical operations. This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide.

- Numeric type requirements
- Numeric arrays
- Generalized Numeric Operations
- C Library

11.1 Numeric type requirements

The complex and valarray components are parameterized by the type of information they contain and manipulate.

A C++ program shall instantiate these components only with a type \texttt{TYPE} that satisfies the following requirements:

Tis not an abstract class (it has no pure virtual member functions);

- \texttt{TYPE} is not a reference type;
- \texttt{TYPE} is not cv-qualified;
- If \texttt{TYPE} is a class, it has a public default constructor;
- If \texttt{TYPE} is a class, it has a public copy constructor with the signature \texttt{TYPE::TYPE(const TYPE&)}
- If \texttt{TYPE} is a class, it has a public destructor;
- If \texttt{TYPE} is a class, it has a public assignment operator whose signature is either \texttt{TYPE& TYPE::operator=(const TYPE&)}
or

\[ \text{TYPE} \& \quad \text{TYPE} \text{::operator=} (\text{TYPE}) \]

- If \text{TYPE} is a class, the assignment operator, copy and default constructors, and destructor shall correspond to each other as far as an initialization of raw storage using the default constructor, followed by assignment, is the equivalent to initialization of raw storage using the copy constructor.
- Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object.
- If \text{TYPE} is a class, it shall not overload unary \text{operator\&}.

If an operation on \text{TYPE} throws an exception then the effects are undefined.

Specific classes member functions or general functions may have other restrictions.

## 11.2 Numeric arrays

The numeric array library consists of several classes and non member operators for the manipulation of array objects.

- Template Class Valarray
- Valarray Non-member Operations
- Class slice
- Template Class Slice_array
- Class Gslice
- Template Class Gslice_array
- Template Class Mask_array
- Template Class Indirect_array

### 11.2.1 Template Class Valarray

The template class \text{valarray} is a single direction smart array with element indexing beginning with the zero element.
11.2.1.1 Constructors

The class `valarray` provides overloaded constructors to create an object of `valarray` in several manners.

```cpp
valarray();
explicit valarray(size_t);
valarray(const T&, size_t);
valarray(const T*, size_t);
valarray(const valarray<T>&);
valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
```

11.2.1.2 Destructor

Removes a `valarray` object from memory.

```cpp
~valarray();
```

11.2.1.3 Assignment Operator

The `valarray` class provides for various means of assignment to an already created object.

```cpp
valarray<T>& operator=(const valarray<T>&);
valarray<T>& operator=(const T&);
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
```

Remarks

A `valarray` object is returned.

**valarray element access**

An index operator is provided for single element access of `valarray` objects.
11.2.1.4 operator[]

This operator provides element access for read and write operations.

```cpp
t T operator[](size_t) const;
T& operator[](size_t);
```

Remarks

A value is returned.

valarray subset operations

An index operator is provided for subset array access.

11.2.1.5 operator[]

The index operator is specialized for subset access to allow both read and write operations.

```cpp
valarray<T> operator[](slice) const;
slice_array<T> operator[](slice);
valarray<T> operator[](const gslice&) const;
gslice_array<T> operator[](const gslice&);
valarray<T> operator[](const valarray<bool>&) const;
mask_array<T> operator[](const valarray<bool>&);
valarray<T> operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);
```

Remarks

These operators return subset of the array. Const-qualified operators return the subset as a new valarray object, and non-const operators return a class template object which has reference semantics to the original array.

11.2.1.6 valarray unary operators

The `valarray` class provides operators for array manipulation.
Operator+
valarray<T> operator+() const;
Returns a valarray sum of \( x + y \);

Operator-
valarray<T> operator-() const;
Returns a valarray result of \( x - y \);

Operator~
valarray<T> operator~() const;
Returns a valarray result of \( x \)~\( y \);

Operator!
valarray<bool> operator!() const;
Returns a bool valarray of \( !x \);

11.2.1.7 Valarray Computed Assignment
The valarray class provides for a means of compound assignment and math operation. A valarray object is returned.

Operator*=
valarray<T>& operator*= (const valarray<T>&);
valarray<T>& operator*= (const T&);
Returns a valarray result of \( x \)*=\( y \);

Operator/= valarray<T>& operator/= (const valarray<T>&);
valarray<T>& operator/= (const T&);
Returns a valarray result of \( x /= y \);

Operator%=
valarray<T>& operator%= (const valarray<T>&);
valarray<T>& operator%= (const T&);
Returns a valarray result of \( x %= y \);

Operator+=
valarray<T>& operator+=(const valarray<T>&);
valarray<T>& operator+=(const T&);
Returns a `valarray` result of `x+=y`;

**Operator-=**

```cpp
valarray<T>& operator=(const valarray<T>&);
valarray<T>& operator=(const T&);
```

Returns a `valarray` result of `x-=y`;

**Operator^=**

```cpp
valarray<T>& operator^=(const valarray<T>&);
valarray<T>& operator^=(const T&);
```

Returns a `valarray` result of `x^=y`;

**Operator&=**

```cpp
valarray<T>& operator&=(const T&);
valarray<T>& operator&=(const valarray<T>&);
```

Returns a `valarray` result of `x&=y`;

**Operator|=**

```cpp
valarray<T>& operator|=(const valarray<T>&);
valarray<T>& operator|=(const T&);
```

Returns a `valarray` result of `x|=y`;

**Operator<<=**

```cpp
valarray<T>& operator<<=(const valarray<T>&);
valarray<T>& operator<<=(const T&);
```

Returns a `valarray` result of `x <<= y`;

**Operator>>=**

```cpp
valarray<T>& operator>>=(const valarray<T>&);
valarray<T>& operator>>=(const T&);
```

Returns a `valarray` result of `x >>= y`;

### 11.2.2 Valarray Member Functions

The `valarray` class provides member functions for array information.

#### 11.2.2.1 size
Tells the size of the array.

```cpp
size_t size() const;
```

**Remarks**
Returns the size of the array.

### 11.2.2.2 sum

Tells the sum of the array elements.

```cpp
T sum() const;
```

**Remarks**
Returns the sum of the array elements.

### 11.2.2.3 min

Tells the smallest element of an array.

```cpp
T min() const;
```

**Remarks**
Returns the smallest element in an array.

### 11.2.2.4 max

Tells the largest element in an array.

```cpp
T max() const;
```

**Remarks**
Returns the largest element in an array.
11.2.2.5  shift

Returns a new array where the elements have been shifted a set amount.

valarray<T> shift(int n) const;

Remarks
Returns the modified array.

11.2.2.6  cshift

A cyclical shift of an array.

valarray<T> cshift(int n) const;

Remarks
Returns the modified array.

11.2.2.7  apply

Processes the elements of an array.

valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;

Remarks
This function "applies" the function specified to all the elements of an array.
Return the modified array.

11.2.2.8  resize

Resizes an array and initializes the elements
void resize(size_t sz, T c = T());

Remarks

If no object is provided the array is initialized with the default constructor.

11.2.3 Valarray Non-member Operations

Non-member operators are provided for manipulation or arrays.

11.2.3.1 Valarray Binary Operators

Non-member valarray operators are provided for the manipulation of arrays.
template<class T> valarray<T> operator^  
(const valarray<T>&, const T&);  
template<class T> valarray<T> operator^  
(const T&, const valarray<T>&);  
template<class T> valarray<T> operator&  
(const valarray<T>&, const T&);  
template<class T> valarray<T> operator&  
(const T&, const valarray<T>&);  
template<class T> valarray<T> operator|  
(const valarray<T>&, const T&);  
template<class T> valarray<T> operator|  
(const T&, const valarray<T>&);  
template<class T> valarray<T> operator<<  
(const valarray<T>&, const T&);  
template<class T> valarray<T> operator<<  
(const T&, const valarray<T>&);  
template<class T> valarray<T> operator>>  
(const valarray<T>&, const T&);  
template<class T> valarray<T> operator>>  
(const T&, const valarray<T>&);

Remarks
Each operator returns an array whose length is equal to the lengths of the argument arrays and initialized with the result of applying the operator.

11.2.3.2 Valarray Logical Operators

The valarray class provides logical operators for the comparison of like arrays.

template<class T> valarray<bool> operator==  
(const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator!=  
(const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator<  
(const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator>  
(const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator<  
(const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator>=  
(const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator&&  
(const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator||  
(const valarray<T>&, const valarray<T>&);

Remarks
All of the logical operators returns a bool array whose length is equal to the length of the array arguments. The elements of the returned array are initialized with a boolean result of the match.
11.2.4 Non-member logical operations

Non-member logical operators are provided to allow for variations of order of the operation.

```cpp
template<class T> valarray<bool> operator==(const valarray&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray&);
template<class T> valarray<bool> operator!=(const valarray&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray&);
template<class T> valarray<bool> operator<(const valarray&, const T&);
template<class T> valarray<bool> operator<(const T&, const valarray&);
template<class T> valarray<bool> operator>=(const valarray&, const T&);
template<class T> valarray<bool> operator>=(const T&, const valarray&);
```

Remarks

The result of these operations is a bool array whose length is equal to the length of the array argument. Each element of the returned array is the result of a logical match.

11.2.4.1 valarray transcendentalss

Trigonometric and exponential functions are provided for the `valarray` classes.

```cpp
template<class T> valarray<T> abs(const valarray<T>&); template<class T> valarray<T> acos(const valarray<T>&); template<class T> valarray<T> asin(const valarray<T>&); template<class T> valarray<T> atan(const valarray<T>&);
```
template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const T&);
template<class T> valarray<T> atan2(const T&, const valarray<T>&);

template<class T> valarray<T> cos(const valarray<T>&);

template<class T> valarray<T> cosh(const valarray<T>&);

template<class T> valarray<T> exp(const valarray<T>&);

template<class T> valarray<T> log(const valarray<T>&);

template<class T> valarray<T> log10(const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const T&);

template<class T> valarray<T> pow(const T&, const valarray<T>&);

template<class T> valarray<T> sin(const valarray<T>&);

template<class T> valarray<T> sinh(const valarray<T>&);

template<class T> valarray<T> sqrt(const valarray<T>&);

template<class T> valarray<T> tan(const valarray<T>&);

template<class T> valarray<T> tanh(const valarray<T>&);

Remarks

A valarray object is returned with the individual elements initialized with the result of the corresponding operation.

11.2.5 Class slice

A slice is a set of indices that have three properties, a starting index, the number of elements and the distance between the elements.

11.2.5.1 Constructors

A constructor is overloaded to initialize an object with values or without values.

slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);
11.2.5.2 slice access functions

The slice class has three member functions.

11.2.5.2.1 start

Start indicates the position where the slice starts.

    size_t start() const;

Remarks
The starting position is returned.

11.2.5.2.2 size

Size indicates the size of the slice.

    size_t size() const;

Remarks
The size of the slice is returned by the size member function.

11.2.5.2.3 stride

The distance between elements is given by the stride function.

    size_t stride() const;

Remarks
The distance between each element is returned by stride.

11.2.6 Template Class Slice_array
The `slice_array` class is a helper class used by the slice subscript operator.

### 11.2.6.1 Constructors

Constructs a `slice_array` object.

```cpp
private:
slice_array();
slice_array(const slice_array&);
```

### 11.2.6.2 Assignment Operator

The assignment operator allows for the initialization of a `slice_array` after construction.

```cpp
void operator=(const valarray<T>&) const;
slice_array& operator=(const slice_array&);
```

### 11.2.6.3 `slice_array computed assignment`

Several compound assignment operators are provided.

```cpp
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

### Remarks

There is no return for the compound operators.

### 11.2.6.4 Slice_array Fill Function

An assignment operation is provided to fill individual elements of the array.
void operator=(const T&);

Remarks
No value is returned.

11.2.7 Class Gslice
A general slice class is provided for multidimensional arrays.

11.2.7.1 Constructors
An overloaded constructor is provided for the creation of a gslice object.

gslice();
gslice(size_t start, const valarray<size_t>& lengths,
const valarray<size_t>& strides);
gslice(const gslice&);

11.2.7.2 Gslice Access Functions
The gslice class provides for access to the start, size and stride of the slice class.

11.2.7.2.1 start
The start function gives the starting position.

size_t start() const;

Remarks
The starting position of the gslice is returned.

11.2.7.2.2 size
The size function returns the number of elements.
valarray<size_t> size() const;

Remarks
The number of elements as a valarray is returned.

11.2.7.2.3 stride

The stride function tells the size of each element.

valarray<size_t> stride() const;

Remarks
The size of the element as a valarray is returned.

11.2.8 Template Class Gslice_array

The `gslice_array` class is a helper class used by the gslice subscript operator.

11.2.8.1 Constructors

An overloaded constructor is provided for the creation of a `gslice_array` object.

`gslice_array();
gslice_array(const gslice_array&);`

11.2.8.2 Assignment Operators

An assignment operator is provided for initializing a `gslice_array` after it has been created.

`void operator=(const valarray<T>&) const;
gslice_array& operator=(const gslice_array&);`

Remarks
A copy of the modified \texttt{gslice_array} is returned for the second assignment operator.

### 11.2.8.3 Gslice_array Computed Assignment

Several compound assignment operators are provided.

```cpp
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

**Remarks**

No return is given for the compound operators.

### 11.2.8.4 Fill Function

An assignment operation is provided to fill individual elements of the array.

```cpp
void operator= (const T&);
```

**Remarks**

There is no return for the fill function.

### 11.2.9 Template Class Mask_array

The \texttt{mask_array} class is a helper class used by the mask subscript operator.

#### 11.2.9.1 Constructors

An overloaded constructor is provided for creating a \texttt{mask_array} object.
11.2.9.2 Assignment Operators

An overloaded assignment operator is provided for assigning values to a mask_array after construction.

```cpp
void operator=(const valarray<T>&) const;
mask_array& operator=(const mask_array&);
```

Remarks

The copy assignment operator returns a mask_array reference.

11.2.9.3 Mask_array Computed Assignment

Several compound assignment operators are provided.

```cpp
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

Remarks

There is no return value for the compound assignment operators.

11.2.9.4 Mask_array Fill Function

An assignment operation is provided to fill individual elements of the array.

```cpp
void operator =(const T&);
```

Remarks
There is no return for the fill function.

11.2.10 Template Class Indirect_array

The `indirect_array` class is a helper class used by the indirect subscript operator.

This template is a helper template used by the indirect subscript operator `indirect_array<T> valarray<T>::operator[](const valarray<size_t>&)`. It has reference semantics to a subset of an array specified by an `indirect_array`.

11.2.10.1 Constructors

An overloaded constructor is provided for creating a `indirect_array` object.

```cpp
indirect_array();
indirect_array(const indirect_array&);
```

11.2.10.2 Assignment Operators

An overloaded assignment operator is provided for assigning values to a `indirect_array` after construction.

```cpp
void operator=(const valarray<T>&) const;
indirect_array& operator=(const indirect_array&);
```

Remarks

The copy assignment operator returns a `indirect_array` reference.

11.2.10.3 Indirect_array Computed Assignment

Several compound assignment operators are provided.

```cpp
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%=(const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
```
Generalized Numeric Operations

void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|=(const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;

Remarks
There is no return value for the compound assignment operators.

11.2.10.4 indirect_array fill function

An assignment operation is provided to fill individual elements of the array.

void operator=(const T&);

Remarks
There is no return for the fill function.

11.3 Generalized Numeric Operations

The standard library provides general algorithms for numeric processing.

11.3.1 Header <numeric>

The header <numeric> includes template functions for generalized numeric processing.

11.3.1.1 accumulate

Accumulate the sum of a sequence.

template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);
template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation binary_op);

Remarks
The sum of the values in a range or the sum of the values after being processed by an operation is returned.

11.3.1.2 inner_product

Computes and returns the value of a product of the values in a range.

\[
\text{template <class InputIterator1, class InputIterator2, class T>}
\]
\[
\text{T inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init);}\
\]
\[
\text{template <class InputIterator1, class InputIterator2, class T, class BinaryOperation1, class BinaryOperation2>}
\]
\[
\text{T inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init,BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);}\
\]

Remarks

The value of the product starting with an initial value in a range is returned. In the function with the operation argument it is the product after the operation is performed.

11.3.1.3 partial_sum

Computes the partial sum of a sequence of numbers.

\[
\text{template <class InputIterator, class OutputIterator>}
\]
\[
\text{OutputIterator partial_sum}
\]
\[
\text{(InputIterator first, InputIterator last, OutputIterator result);}\
\]
\[
\text{template <class InputIterator, class OutputIterator, class BinaryOperation>}
\]
\[
\text{OutputIterator partial_sum}
\]
\[
\text{(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation binary_op);}\
\]

The first computes the partial sum and sends it to the output iterator argument.

\[
x, y, z
\]
\[
x, x+y, y+z.
\]

The second form computes according to the operational argument and sends it to the output iterator argument. For example if the operational argument was a multiplication operation

\[
x, y, z
\]
\[
x, x*y, y*z
\]
Remarks
The range as the result plus the last minus the first.

11.3.1.4 adjacent_difference

Computed the adjacent difference in a sequence of numbers.

```cpp
template <class InputIterator, 
          class OutputIterator>
OutputIterator adjacent_difference
(InputIterator first, InputIterator last,
 OutputIterator result);

template <class InputIterator, 
          class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference
(InputIterator first, InputIterator last,
 OutputIterator result,
 BinaryOperation binary_op);
```

The first computes the adjacent difference and sends it to the output iterator argument.

\[
x, y, z
\]

\[
x, y-x, z-y.
\]

The second form computes according to the operational argument and sends it to the output iterator argument. For example if the operational argument was a division operation

\[
x, y, z
\]

\[
x, y/x, z/y
\]

Remarks
The range as the result plus the last minus the first.

11.4 C Library

The standard provides for the math functions included in the standard C library with some overloading for various types.
11.4.1 <cmath>

The contents of the <cmath> headers is the same as the Standard C library headers <math.h> with the addition to the double versions of the math functions in <cmath>, C++ adds float and long double overloaded versions of some functions, with the same semantics.

11.4.2 <cstdlib>

The contents of the <cstdlib> headers is the same as the Standard C library headers <stdlib.h>. In addition to the int versions of certain math functions in <cstdlib>, C++ adds long overloaded versions of some functions, with the same semantics.

Listing: The Added C++ Signatures in Cstdlib and Cmath

```c
long double abs (long double);
long double acos (long double);
long double asin (long double);
long double atan (long double);
long double atan2(long double, long double);
long double ceil (long double);
long double cos (long double);
long double cosh (long double);
long double exp (long double);
long double fabs (long double);
long double floor(long double);
long double fmod (long double, long double);
long double frexp(long double, int*);
long double ldexp(long double, int);
long double log (long double);
long double log10(long double);
long double modf (long double, long double*);
long double pow (long double, long double);
long double pow (long double, int);
long double sin (long double);
long double sinh (long double);
long double sqrt (long double);
long double tan (long double);
long double tanh (long double);
float abs (float);
float acos (float);
float asin (float);
float atan (float);
float atan2(float, float);
float ceil (float);
float cos (float);
float cosh (float);
float exp (float);
float fabs (float);
float floor(float);
float fmod (float, float);
float frexp(float, int*);
float ldexp(float, int);
float log (float);
float log10(float);
```
C Library

float modf (float, float*);
float pow (float, float);
float pow (float, int);
float sin (float);
float sinh (float);
float sqrt (float);
float tan (float);
double abs(double);
double pow(double, int);
Chapter 12
Complex Class

The header `<complex>` defines a template class, and facilities for representing and manipulating complex numbers.

The header `<complex>` defines classes, operators, and functions for representing and manipulating complex numbers.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Header complex shows the complex header class declarations
- Complex Specializations lists the float, double and long double specializations
- Complex Template Class is a template class for complex numbers.

12.1 Header complex

The header `<complex>` defines classes, operators, and functions for representing and manipulating complex numbers.

12.1.1 _EWL_CX_LIMITED_RANGE

This flag effects the * and / operators of complex.

When defined, the "normal" formulas for multiplication and division are used. They may execute faster on some machines. However, infinities will not be properly calculated, and there is more roundoff error potential.
If the flag is undefined (default), then more complicated algorithms (from the C standard) are used which have better overflow and underflow characteristics and properly propagate infinity. Flipping this switch requires recompilation of the C++ library.

**NOTE**

It is recommended that `ansi_prefix.xxx.h` is the place to define this flag if you want the simpler and faster multiplication and division algorithms.

### 12.1.2 Header `<complex>` forward declarations

The complex class has forward declarations.

- `template<class T> class complex;`
- `template<> class complex<float>;`
- `template<> class complex<double>;`
- `template<> class complex<long double>;`

### 12.2 Complex Specializations

The standard specializes the template complex class for float, double and long double types.

### 12.3 Complex Template Class

The template class `complex` contains Cartesian components `real` and `imag` for a complex number.

**Remarks**

The effect of instantiating the template complex for any type other than float, double or long double is unspecified. If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

The complex class consists of:

- **Constructors and Assignments**
- **Complex Member Functions**
12.3.1 Constructors and Assignments

Constructor, destructor and assignment operators and functions.

12.3.1.1 Constructors

Construct an object of a complex class.

```cpp
complex(const T& re = T(), const T& im = T());
complex(const complex&);
template<class X> complex(const complex<X>&);
```

Remarks

After construction real equal re and imag equals im.

AssignmentOperator

An assignment operator for complex classes.

```cpp
complex<T>& operator= (const T&);
complex& operator= (const complex&);

template<class X> complex<T>& operator= (const complex<X>&);
```

Remarks

Assigns a floating point type to the Cartesian complex class.

12.3.2 Complex Member Functions

There are two public member functions, real and imag.

There are two public member functions.
12.3.2.1 real

Retrieves the real component.

\[ T \text{ real()} \text{ const}; \]

12.3.2.2 imag

Retrieves the imag component.

\[ T \text{ imag()} \text{ const}; \]

12.3.3 Complex Class Operators

Several assignment operators are overloaded for the complex class manipulations.

operator+=

Adds and assigns to a complex class.

\[
\begin{align*}
&\text{complex}<T>& \text{ operator+=}(\text{const } T&)\
&\text{template<class } X> \text{ complex}<T>& \text{ operator+=}
&\quad(\text{const } \text{ complex}<X>&);
\end{align*}
\]

Remarks

The first operator with a scalar argument adds the scalar value of the right hand side to the real component and stores the result in the object. The imaginary component is left alone.

The second operator with a complex type, adds the complex value of the right hand side to the object and stores the result in the object.

The this pointer is returned.
operator-=
Subtracts and assigns from a complex class.

complex<T>& operator-=(const T&);
template<class X> complex<T>& operator-=(const complex<X>&);

Remarks
The first operator with a scalar argument subtracts the scalar value of the right hand side
from the real component and stores the result in the object. The imaginary component is
left alone.

The second operator with a complex type, subtracts the complex value of the right hand
side from the object and stores the result in the object.

The this pointer is returned.

operator*=
Multiplies by and assigns to a complex class.

complex<T>& operator*=(const T&);
template<class X> complex<T>& operator*=(const complex<X>&);

Remarks
The first operator with a scalar argument multiplies the scalar value of the right hand side
to class object and stores result in the object.

The second operator with a complex type, multiplies the complex value of the right hand
side to the object and stores the result in the object.

The this pointer is returned.

operator/= 
Divides by and assigns to a complex class.

complex<T>& operator/= (const T&);
template<class X> complex<T>& operator/= (const complex<X>&);

Remarks
The first operator with a scalar argument divides the scalar value of the right hand side to
class object and stores result in the object.
The second operator with a complex type, divides the complex value of the right hand side into the object and stores the result in the object.

The `this` pointer is returned.

### 12.3.4 Overloaded Operators and Functions

There are several non member functions and overloaded operators in the complex class library.

- Overloaded Complex Operators
- Complex Value Operations
- Complex Transcendentals

#### 12.3.4.1 Overloaded Complex Operators

The overloaded complex operators consists of:

**operator+**

Adds to the complex class.

```cpp
template<class T> complex<T> operator+ 
    const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator+ 
    (const complex<T>&, const T&);  
template<class T> complex<T> operator+ 
    (const T&, const complex<T>&);  
template<class T> complex<T> operator+ 
    (const complex<T>&);  
```

**Remarks**

The addition performs a += operation.

Returns the complex class after the addition.

**operator-**

Subtracts from the complex class.

```cpp
template<class T> complex<T> operator- 
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator- 
    (const complex<T>&, const T&);  
template<class T> complex<T> operator- 
    (const T&, const complex<T>&);  
template<class T> complex<T> operator- 
    (const complex<T>&);  
```
template<class T> complex<T> operator- (const complex<T>&);

Remarks
The subtraction performs a -= operation.
Returns the complex class after the Subtraction.

operator*
Multiplies the complex class.

template<class T> complex<T> operator* (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator* (const complex<T>&, const T&);
template<class T> complex<T> operator* (const T&, const complex<T>&);

Remarks
The multiplication performs a *= operation.
Returns the complex class after the multiplication.

operator/
Divides from the complex class.

template<class T> complex<T> operator/ (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/ (const complex<T>&, const T&);
template<class T> complex<T> operator/ (const T&, const complex<T>&);

Remarks
The division performs a /= operation.
Returns the complex class after the division.

operator==
A boolean equality comparison.

template<class T> bool operator== (const complex<T>&, const complex<T>&);
template<class T> bool operator== (const complex<T>&, const T&);
template<class T> bool operator== (const T&, const complex<T>&);
Remarks
Returns true if the real and imaginary components are equal.

\texttt{operator!=}

A boolean non equality comparison.

\begin{verbatim}
template<class T> bool operator!=
  (const complex<T>&, const complex<T>&);
template<class T> bool operator!=
  (const complex<T>&, const T&);
template<class T> bool operator!=
  (const T&, const complex<T>&);
\end{verbatim}

Remarks
Returns true if the real or the imaginary components are not equal.

\texttt{operator>>}

Extracts a complex type from a stream.

\begin{verbatim}
template<class T, class charT, class traits>
  basic_istream<charT, traits>& operator>>
  (basic_istream<charT, traits>&, complex<T>&);
\end{verbatim}

Remarks
Extracts in the form of u, (u), or (u,v) where u is the real part and v is the imaginary part. Any failure in extraction will set the failbit and result in undefined behavior.

\texttt{operator<<}

Inserts a complex number into a stream.

\begin{verbatim}
template<class T, class charT, class traits>
  basic_ostream<charT, traits>& operator<<
  (basic_ostream<charT, traits>&, const complex<T>&);
\end{verbatim}

12.3.5 Complex Value Operations

This section lists the complex value operations.

- \texttt{real}
- \texttt{imag}
- \texttt{abs}
12.3.5.1  real

Retrieves the real component of a complex class.

    template<class T> T real(const complex<T>&);

**Remarks**

Returns the real component of the argument.

12.3.5.2  imag

Retrieves the imaginary component of a complex class.

    template<class T> T imag(const complex<T>&);

**Remarks**

Returns the imaginary component of the argument.

12.3.5.3  abs

Determines the absolute value of a complex class.

    template<class T> T abs(const complex<T>&);

**Remarks**

Returns the absolute value of the complex class argument.
12.3.5.4  arg

Determines the phase angle.

\[
\text{template<class } T\text{> } T \arg(const \text{complex}<T>&);
\]

**Remarks**

Returns the phase angle of the complex class argument or \( \text{atan2(\text{imag(x)}, \text{real(x)})} \).

12.3.5.5  norm

Determines the squared magnitude.

\[
\text{template<class } T\text{> } T \text{norm(const complex}<T>&);
\]

**Remarks**

The squared magnitude of the complex class.

12.3.5.6  conj

Determines the complex conjugate.

\[
\text{template<class } T\text{> complex}<T> \text{conj(const complex}<T>&);
\]

**Remarks**

Returns the complex conjugate of the complex class argument.

12.3.5.7  polar

Determines the polar coordinates.

\[
\text{template<class } T\text{> complex}<T> \text{polar(const } T\text{&}, \text{const } T\text{&);}
\]
Remarks
Returns the complex value corresponding to a complex number whose magnitude is the first argument and whose phase angle is the second argument.

12.3.6 Complex Transcendentals
This section lists the complex transcendentals.

- cos
- cosh
- exp
- log
- log10
- pow
- sin
- sinh
- sqrt
- tan
- tanh

12.3.6.1 cos
Determines the cosine.

    template<class T> complex<T> cos (const complex<T>&);

Remarks
Returns the cosine of the complex class argument.

12.3.6.2 cosh
Determines the hyperbolic cosine.

    template<class T> complex<T> cosh (const complex<T>&);
Remarks
Returns the hyperbolic cosine of the complex class argument.

12.3.6.3  exp

Determines the exponential.

    template<class T> complex<T> exp (const complex<T>&);

Remarks
Returns the base exponential of the complex class argument.

12.3.6.4  log

Determines the natural base logarithm.

    template<class T>
    complex<T> log (const complex<T>&);

Remarks
Returns the natural base logarithm of the complex class argument, in the range of a strip mathematically unbounded along the real axis and in the interval of \([i\pi, i\pi]\) along the imaginary axis. The argument is a negative real number, \(\text{imag}(\log(cpx))\), is \(\pi\).

12.3.6.5  log10

Determines the logarithm to base ten.

    template<class T>
    complex<T> log10(const complex<T>&);

Remarks
Returns the logarithm base(10) of the argument \(cpx\) defined as \(\log(cpx)/\log(10)\).
12.3.6.6 pow

Raises the complex class to a set power.

```
template<class T> complex<T> pow(const complex<T>&, int);
template<class T> complex<T> pow(const complex<T>&, const T&);
template<class T> complex<T> pow(const complex<T>&, const complex<T>&);
```

Remarks

Returns the complex class raised to the power of second argument defined as the exponent of the second argument times the log of the first argument.

The value for \( \text{pow}(0,0) \) will return (nan, nan).

12.3.6.7 sin

Determines the sine.

```
template<class T>
complex<T> sin (const complex<T>&);
```

Remarks

Returns the sine of the complex class argument.

12.3.6.8 sinh

Determines the hyperbolic sine.

```
template<class T>
complex<T> sinh (const complex<T>&);
```

Remarks

Returns the hyperbolic sine of the complex class argument.
12.3.6.9   sqrt

Determines the square root.

```cpp
    template<class T>
    complex<T> sqrt (const complex<T>&);
```

**Remarks**

Returns the square root of the complex class argument in the range of right half plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis.

12.3.6.10  tan

Determines the tangent.

```cpp
    template<class T>
    complex<T> tan (const complex<T>&);
```

**Remarks**

Returns the tangent of the complex class argument.

12.3.6.11  tanh

Determines the hyperbolic tangent.

```cpp
    template<class T>
    complex<T> tanh (const complex<T>&);
```

**Remarks**

Returns the hyperbolic tangent of the complex class argument.
Chapter 13
Input and Output Library

A set of components that C++ programs may use to perform input/output operations.

This chapter is constructed in the following subsections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Input and Output Library Summary
- Iostreams requirements

13.1 Input and Output Library Summary

This section lists the headers available in the Input/Output Library.

<table>
<thead>
<tr>
<th>Include</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;iosfwd&gt;</td>
<td>Forward declarations</td>
</tr>
<tr>
<td>&lt;iostream&gt;</td>
<td>Standard iostream objects</td>
</tr>
<tr>
<td>&lt;ios&gt;</td>
<td>iostream base classes</td>
</tr>
<tr>
<td>&lt;streambuf&gt;</td>
<td>Stream buffers</td>
</tr>
<tr>
<td>&lt;istream&gt;</td>
<td>Formatting and manipulators</td>
</tr>
<tr>
<td>&lt;ostream&gt;</td>
<td>Output streams</td>
</tr>
<tr>
<td>&lt;iomanip&gt;</td>
<td>Input and output manipulators</td>
</tr>
<tr>
<td>&lt;sstream&gt;</td>
<td>String streams</td>
</tr>
<tr>
<td>&lt;cstdlib&gt;</td>
<td>Standard C utilities</td>
</tr>
<tr>
<td>&lt;fstream&gt;</td>
<td>File Streams</td>
</tr>
<tr>
<td>&lt;cstdio&gt;</td>
<td>Standard C input and output support</td>
</tr>
<tr>
<td>&lt;cwchar&gt;</td>
<td>Standard C wide characters support</td>
</tr>
</tbody>
</table>
13.2 Iostreams requirements

The EWL C Library is not specifically required for EWL C++ input and output operations.

Topics in this section are:

- Definitions
- Type requirements
- Type SZ_T

13.2.1 Definitions

Additional definitions are:

- character - A unit that can represent text
- character container type - A class or type used to represent a character.
- iostream class templates - Templates that take two arguments: charT and traits. The argument charT is a character container type. The argument traits is a structure which defines characteristics and functions of the charT type.
- narrow-oriented iostream classes - These classes are template instantiation classes. The traditional iostream classes are narrow-oriented iostream classes.
- wide-oriented iostream classes - These classes are template instantiation classes. They are used for the character container class wchar_t.
- repositionable streams and arbitrary-positional streams - A repositionable stream can seek to only a previously encountered position. An arbitrary-positional stream can seek to an integral position within the length of the stream.

13.2.2 Type requirements

Several types are required by the standards, they are consolidated in the strings library.

13.2.3 Type SZ_T

A type that represents one of the signed basic integral types. It is used to represent the number of characters transferred in an input/output operation or the size of the input/output buffers.
Chapter 14
Forward Declarations

The header `<iosfwd>` is used for forward declarations of template classes.

The non-standard header `<stringfwd>` is used for forward declarations of string class objects.

14.1 The Streams and String Forward Declarations

The ANSI/ISO standard calls for forward declarations of input and output streams for basic input and output.

This is for both normal and wide character formats.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

14.2 Header iosfwd

The header `<iosfwd>` is used for forward declarations of template classes.

The template class `basic_ios<charT, traits>` serves as a base class for class `basic_istream` and `basic_ostream`.

The class `ios` is an instantiation of `basic_ios` specialized by the type `char`.

The class `wios` is an instantiation of `basic_ios` specialized by the type `wchar_t`.

14.3 Header stringfwd
This non-standard header can be used to forward declare basic_string (much like <iosfwd> forward declares streams). There is also a <stringfwd.h> that forward declares basic_string and places it into the global namespace.

**NOTE**

The header <stringfwd> is a non-standard header.

**Listing: Header <stringfwd> Synopsis**

```cpp
namespace std { // Optional
    template <class T> class allocator;
    template<class charT> struct char_traits;
    template <class charT, class traits, class Allocator> class basic_string;
    typedef basic_string <char, char_traits<char>, allocator<char> > string;
    typedef basic_string <wchar_t, char_traits<wchar_t>, allocator<wchar_t> > wstring;
}
```

Including <stringfwd> allows you to use a string object.

**Listing: Example of <stringfwd> Inclusion of std::string**

```cpp
#include <stringfwd>
#include <string>

int main()
{
    string a("Hi");     // no std:: required
    return 0;
}
```

The headers <stringfwd.h> and <string> can be used in combination to place string into the global namespace, much like is done with other <name.h> headers. The header <string.h> does not work because that is a standard C header.

**Listing: Example of Stringfwd usage**

```cpp
#include <stringfwd.h>
#include <string>

int main()
{
    string a("Hi");     // no std:: required
    return 0;
}
```
Chapter 15
Iostream Objects

The include header `<iostream>` declares input and output stream objects. The declared objects are associated with the standard C streams provided for by the functions in `<cstdio>`. This chapter uses the ISO (International Organization for Standardization) C++ Standard as a guide.

15.1 Header iostream

The header `<iostream>` declares standard input and output objects in namespace `std`.

Listing: Iostream input and output objects

```cpp
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;
extern wistream wcin;
extern wostream wcout;
extern wostream cerr;
extern wostream wclog;
```

15.1.1 Stream Buffering

All streams are buffered (by default) except `cerr` and `wcerr`.
NOTE
You can change the buffering characteristic of a stream with:

```cpp
cout.setf(ios_base::unitbuf);
```

or

```cpp
cerr.unsetf(ios_base::unitbuf);
```

**Tip**
Do not include `<iostream>` unless needed. Including and not using `<iostream>` will add code size. If you really need it, consider including only `<niostream>` instead. This will instantiate only the narrow console streams, not the wide ones.

### 15.2 The Standard Input and Output Stream Library

The ANSI/ISO standard calls for predetermined objects for standard input, output, logging and error reporting.

This is initialized for normal and wide character formats.

- **Narrow stream objects**
- **Wide stream objects**

#### 15.2.1 Narrow stream objects

Narrow stream objects provide unbuffered input and output associated with standard input and output declared in `<cstdio>`.

##### 15.2.1.1 istream cin

An unbuffered input stream.

```cpp
istream cin;
```
Remarks

The object `cin` controls input from an unbuffered stream buffer associated with `stdin` declared in `<cstdio>`. After `cin` is initialized `cin.tie()` returns `cout`.

Returns an istream object.

15.2.1.2  ostream `cout`

An unbuffered output stream.

    ostream cout;

Remarks

The object `cout` controls output to an unbuffered stream buffer associated with `stdout` declared in `<cstdio>`.

15.2.1.3  ostream `cerr`

Controls output to an unbuffered stream.

    ostream cerr;

Remarks

The object `cerr` controls output to an unbuffered stream buffer associated with `stderr` declared in `<cstdio>`. After `err` is initialized, `err.flags()` and `unitbuf` is nonzero.

15.2.1.4  ostream `clog`

Controls output to a stream buffer.

    ostream clog;

Remarks
The object `clog` controls output to a stream buffer associated with `cerr` declared in `<cstdio>`.

### 15.2.2 Wide stream objects

Wide stream objects provide unbuffered input and output associated with standard input and output declared in `<cstdio>`.

#### 15.2.2.1 wistream wcin

An unbuffered wide input stream.

```cpp
wistream wcin;
```

**Remarks**

The object `wcin` controls input from an unbuffered wide stream buffer associated with `stdin` declared in `<cstdio>`. After `wcin` is initialized `wcin.tie()` returns `wout`.

#### 15.2.2.2 wostream wcout

An unbuffered wide output stream.

```cpp
wostream wcout;
```

**Remarks**

The object `wcout` controls output to an unbuffered wide stream buffer associated with `stdout` declared in `<cstdio>`.

#### 15.2.2.3 wostream wcerr

Controls output to an unbuffered wide stream.
wostream wcerr;

Remarks
The object werr controls output to an unbuffered wide stream buffer associated with stderr declared in <cstdio>. After werr is initialized, werr.flags() and unitbuf is nonzero.

15.2.2.4 wostream wlcog

Controls output to a wide stream buffer.

wostream wclog;

Remarks
The object wlog controls output to a wide stream buffer associated with cerr declared in <cstdio>.
Chapter 16
Iostreams Base Classes

The include header `<ios>` contains the basic class definitions, types, and enumerations necessary for input and output stream reading, writing, and other manipulations.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Header `ios`
- Typedef Declarations
- Class `ios_base`
- Template class `basic_ios`
- `ios_base` manipulators

16.1 Header `ios`

The header file `<ios>` provides for implementation of stream objects for standard input and output.

16.1.1 Template Class `fpos`

The template class `fpos<stateT>` is a class used for specifying file position information. The template parameter corresponds to the type needed to hold state information in a multi-byte sequence (typically `mbstate_t` from `<cwchar>`). `fpos` is essentially a wrapper for whatever mechanisms are necessary to hold a stream position (and multi-byte state). In fact the standard stream position typedefs are defined in terms of `fpos`:
typedef fpos<mbstate_t> streampos;
typedef fpos<mbstate_t> wstreampos;

The template class fpos is typically used in the istream and ostream classes in calls involving file position such as tellg, tellp, seekg and seekp. Though in these classes the fpos is typedef'd to pos_type, and can be changed to a custom implementation by specifying a traits class in the stream's template parameters.

16.2 Typedef Declarations

The following typedef's are defined in the class ios_base.

typedef long streamoff;
typedef long streamsize;

16.3 Class ios_base

A base class for input and output stream mechanisms.

The prototype is listed below. Additional topics in this section are:

- Typedef Declarations
- Class ios_base::failure
- Type fmtflags
- Type iostate
- Type openmode
- Type seekdir
- Class Init
- Class Init Constructor
- ios_base fmtflags state functions
- ios_base locale functions
- ios_base storage function
- ios_base

The ios_base class is a base class and includes many enumerations and mechanisms necessary for input and output operations.
16.3.1 Typedef Declarations
No types are specified in the current standards.

16.3.2 Class ios_base::failure
Defines a base class for types of objects thrown as exceptions.

16.3.2.1 failure
Construct a class failure.

    explicit failure(const string& msg);

Remarks
The function failure() constructs a class failure initializing with exception(msg).

16.3.2.2 failure::what
To return the exception message.

    const char *what() const;

Remarks
The function what() is used to deliver the msg.str().
Returns the message with which the exception was created.

16.3.3 Type fmtflags
An enumeration used to set various formatting flags for reading and writing of streams.
Table 16-1. Format Flags Enumerations

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effects when set</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolalpha</td>
<td>insert or extract bool type in alphabetic form</td>
</tr>
<tr>
<td>dec</td>
<td>decimal output</td>
</tr>
<tr>
<td>fixed</td>
<td>when set shows floating point numbers in normal manner, six decimal places is default</td>
</tr>
<tr>
<td>hex</td>
<td>hexadecimal output</td>
</tr>
<tr>
<td>oct</td>
<td>octal output</td>
</tr>
<tr>
<td>left</td>
<td>left justified</td>
</tr>
<tr>
<td>right</td>
<td>right justified</td>
</tr>
<tr>
<td>internal</td>
<td>pad a field between signs or base characters</td>
</tr>
<tr>
<td>scientific</td>
<td>show scientific notation for floating point numbers</td>
</tr>
<tr>
<td>showbase</td>
<td>shows the bases numeric values</td>
</tr>
<tr>
<td>showpoint</td>
<td>shows the decimal point and trailing zeros</td>
</tr>
<tr>
<td>showpos</td>
<td>shows the leading plus sign for positive numbers</td>
</tr>
<tr>
<td>skipws</td>
<td>skip leading white spaces with input</td>
</tr>
<tr>
<td>unitbuf</td>
<td>buffer the output and flush after insertion operation</td>
</tr>
<tr>
<td>uppercase</td>
<td>show the scientific notation, x or o in uppercase</td>
</tr>
</tbody>
</table>

Table 16-2. Format flag field constants

<table>
<thead>
<tr>
<th>Constants</th>
<th>Allowable values</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustfield</td>
<td>left</td>
</tr>
<tr>
<td>basefield</td>
<td>dec</td>
</tr>
<tr>
<td>floatfield</td>
<td>scientific</td>
</tr>
</tbody>
</table>

Listing: Example of ios format flags usage

see basic_ios::setf() and basic_ios::unsetf()

16.3.4 Type iostate

An enumeration that is used to define the various states of a stream.

Table 16-3. Enumeration iostate

<table>
<thead>
<tr>
<th>Flags</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>goodbit</td>
<td>True when all of badbit, eofbit and failbit are false.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 16-3. Enumeration iostate (continued)

<table>
<thead>
<tr>
<th>Flags</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>badbit</td>
<td>True when the stream is in an irrecoverable error state (such as failure due to lack of memory)</td>
</tr>
<tr>
<td>failbit</td>
<td>True when a read or write has failed for any reason. (This can happen for example when the input reads a character while attempting to read an integer.)</td>
</tr>
<tr>
<td>eofbit</td>
<td>True when the end of the stream has been detected. Note that eofbit can be set during a read, and yet the read may still succeed (failbit not set). (This can happen for example when an integer is the last character in a file.)</td>
</tr>
</tbody>
</table>

Note: see variance from AT&T standard.

For an example of ios iostate flags usage refer to `basic_ios::setstate()` and `basic_ios::rdstate()`

### 16.3.5 Type openmode

An enumeration that is used to specify various file opening modes.

Table 16-4. Enumeration openmode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>Start the read or write at end of the file</td>
</tr>
<tr>
<td>ate</td>
<td>Start the read or write immediately at the end</td>
</tr>
<tr>
<td>binary</td>
<td>binary file</td>
</tr>
<tr>
<td>in</td>
<td>Start the read at end of the stream</td>
</tr>
<tr>
<td>out</td>
<td>Start the write at the beginning of the stream</td>
</tr>
<tr>
<td>trunc</td>
<td>Start the read or write at the beginning of the stream</td>
</tr>
</tbody>
</table>

### 16.3.6 Type seekdir

An enumeration to position a pointer to a specific place in a file stream.
Table 16-5. Enumeration seekdir

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>beg</td>
<td>Beginning of stream</td>
</tr>
<tr>
<td>cur</td>
<td>Current position of stream</td>
</tr>
<tr>
<td>end</td>
<td>End of stream</td>
</tr>
</tbody>
</table>

For an example of ios seekdir usage refer to streambuf::pubseekoff

16.3.7 Class Init

An object that associates `<iostream>` object buffers with standard stream declared in `<cstdio>`.

16.3.7.1 Class Init Constructor

To construct an object of class Init;

    Init();

Remarks

The default constructor `Init()` constructs an object of class `Init`. If `init_cnt` is zero the function stores the value one and constructs `cin`, `cout`, `cerr`, `clog`, `wcin`, `wout`, `werr` and `wlog`. In any case the constructor then adds one to `init_cnt`.

16.3.7.2 Destructor

    ~Init();

Remarks

The destructor subtracts one from `init_cnt` and if the result is one calls `cout.flush()`, `cerr.flush()` and `clog.flush()`.
16.3.8  **ios_base fmtflags state functions**

To set the state of the ios_base format flags.

16.3.8.1  **flags**

To alter formatting flags using a mask.

```cpp
fmtflags flags() const
fmtflags flags(fmtflags)
```

**Remarks**

Use `flags()` when you would like to use a mask of several flags, or would like to save the current format configuration. The return value of `flags()` returns the current `fmtflags`. The overloaded `flags(fmtflags)` alters the format flags but will return the value prior to the flags being changed.

The `fmtflags` type before alterations.

See `ios` enumerators for a list of `fmtflags`.

**SeeAlso:**

`setiosflags()` and `resetiosflags()`

**Listing: Example of flags() usage:**

```cpp
#include <iostream>
// showf() displays flag settings
void showf();
int main()
{
    using namespace std;
    showf(); // show format flags
    cout << "press enter to continue" << endl;
    cin.get();
    cout.setf(ios::right|ios::showpoint|ios::fixed);
    showf();
    return 0;
}
```
// showf() displays flag settings
void showf()
{
using namespace std;

    char fflags[12] = {
        "boolalpha",
        "dec",
        "fixed",
        "hex",
        "internal",
        "left",
        "oct",
        "right",
        "scientific",
        "showbase",
        "showpoint",
        "showpos",
        "skipws",
        "unitbuf",
        "uppercase"
    };

    long f = cout.flags();    // get flag settings

    cout.width(9); // for demonstration

    // check each flag
    for(long i=1, j =0; i<=0x4000; i = i<<1, j++)
    {
        cout.width(10); // for demonstration
        if(i & f)
            cout << fflags[j] << " is on 
";
        else
            cout << fflags[j] << " is off 
";
    }
    cout << "
";
}
Result:

boolalpha    is off
dec         is on
fixed       is off
hex          is off
internal    is off
left        is off
oct          is off
right       is off
scientific is off
showbase    is off
showpoint   is off
showpos     is off
skipws      is on
unitbuf     is off
uppercase   is off
press enter to continue
boolalpha is off
dec is on
hex is off
internal is off
left is off
oct is off
right is on
scientific is off
showbase is off
showpoint is on
showpos is off
skipws is on
unitbuf is off
uppercase is off

16.3.8.2  setf

Set the stream format flags.

fmtflags setf(fmtflags)
fmtflags setf(fmtflags, fmtflags)

Remarks

You should use the function setf() to set the formatting flags for input/output. It is overloaded. The single argument form of setf() sets the flags in the mask. The two argument form of setf() clears the flags in the first argument before setting the flags with the second argument.

type basic_ios::fmtflags is returned.

Listing: Example of setf() usage:

#include <iostream>
int main()
using namespace std;

double d = 10.01;
cout.setf(ios::showpos | ios::showpoint);
cout << d << endl;
cout.setf(ios::showpoint, ios::showpos | ios::showpoint);
cout << d << endl;
return 0;

Result:

+10.01
10.01

16.3.8.3  unsetf

To un-set previously set formatting flags.

void unsetf(fmtflags)

Remarks

Use the unsetf() function to reset any format flags to a previous condition. You would normally store the return value of setf() in order to achieve this task.

There is no return.

Listing: Example of unsetf() usage:

#include <iostream>
int main()
{
using namespace std;

double d = 10.01;
cout.setf(ios::showpos | ios::showpoint);
cout << d << endl;
cout.unsetf(ios::showpoint);
cout << d << endl;
return 0;
Result:

+10.01
+10.01

### 16.3.8.4 precision

Set and return the current format precision.

```cpp
streamsize precision() const
streamsize precision(streamsize prec)
```

**Remarks**

Use the `precision()` function with floating point numbers to limit the number of digits in the output. You may use `precision()` with scientific or non-scientific floating point numbers. You may use the overloaded `precision()` to retrieve the current precision that is set.

With the flag `ios::floatfield` set, the number in `precision` refers to the total number of significant digits generated. If the settings are for either `ios::scientific` or `ios::fixed` then the precision refers to the number of digits after the decimal place.

This means that `ios::scientific` will have one more significant digit than `ios::floatfield`, and `ios::fixed` will have a varying number of digits.

**SeeAlso**

`setprecision()`

**Listing: Example of precision() usage:**

```cpp
#include <iostream>
#include <cmath>

const double pi = 4 * std::atan(1.0);

int main()
{
    using namespace std;

    double TenPi = 10*pi;

    cout.precision(5);
```
cout.unsetf(ios::floatfield);
cout << "floatfield:\t" << TenPi << endl;
cout.setf(ios::scientific, ios::floatfield);
cout << "scientific:\t" << TenPi << endl;
cout.setf(ios::fixed, ios::floatfield);
cout << "fixed:\t\t" << TenPi << endl;
return 0;
}

Result:

floatfield: 31.416
scientific: 3.14159e+01
fixed:         31.41593

16.3.8.5 width

To set the width of the output field.

streamsize width() const
streamsize width(streamsize wide)

Remarks

Use the width() function to set the field size for output. The function is overloaded to return just the current width setting if there is no parameter or to store and then return the previous setting before changing the fields width to the new parameter.

Width is the one and only modifier that is not sticky and needs to be reset with each use. Width is reset to width(0) after each use.

The previous width setting is returned.

Listing: Example of width() usage:

```cpp
#include <iostream>
int main()
{
    using namespace std;
    int width;
    cout.width(8);
```
width = cout.width();
cout.fill('*');
cout << "Hi!" << '\n';
// reset to left justified blank filler
cout<< "Hi!" << '\n';
cout.width(width);
cout<< "Hi!" << endl;
return 0;
}

Result:

Hi!*****
Hi!
Hi!*****

16.3.9  ios_base locale functions
Sets the locale for input output operations.

16.3.9.1  imbue
Stores a value representing the locale.

locale imbue(const locale loc);

Remarks
The precondition of the argument loc is equal to getloc().
The previous value of getloc().

16.3.9.2  getloc
Determines the imbed locale for input output operations.

locale getloc() const;
Remarks
Returns the global C++ locale if no locale has been imbued. Otherwise it returns the locale of the input and output operations.

16.3.10  ios_base storage function

To allocate storage pointers.

16.3.10.1  xalloc

Allocation function.

static int xalloc()

Remarks
Returns index++.

16.3.10.2  iword

Allocates an array of int and stores a pointer.

long& iword(int idx);

Remarks
If iarray is a null pointer, allocate an array and store a pointer to the first element. The function extends the array as necessary to include iarray[idx]. Each newly allocated element of the array is initialized to zero.

The reference returned is invalid after any other operation on the object.

Returns iarray[idx]
16.3.10.3 pword

Allocate an array of pointers.

    void*& pword(int idx);

Remarks

If parray is a null pointer, allocates an array of void pointers. Then extends parray as necessary to include the element parray[idx].

The reference returned is invalid after any other operation on the object.

Returns parray[idx].

16.3.10.4 register_callback

Registers functions when an event occurs.

    void register_callback
        (event_callback fn,
         int index);

Remarks

Registers the pair (fn, index) such that during calls to imbue(), copyfmt() or ~ios_base() the function fn is called with argument index. Registered functions are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

Identical pairs are not merged and a function registered twice will be called twice.

16.3.10.5 sync_with_stdio

Synchronizes stream input output with 'C' input and output functions.

    static bool sync_with_stdio(bool sync = true);

Remarks

Is not supported in the EWL.
Always returns `true` indicating that the EWL streams are always synchronized with the C streams.

### 16.3.11 ios_base

ios_base

#### 16.3.11.1 ios_base Constructor

Construct an object of class ios_base

```cpp
protected:
    ios_base();
```

**Remarks**

The `ios_base` constructor is protected so it may only be derived from. The values of the `ios_base` members are undermined.

#### 16.3.11.2 ios_base Destructor

Destruct an object of class ios_base

```cpp
~ios_base();
```

**Remarks**

Calls registered callbacks and destroys an object of class ios_base.

### 16.4 Template class basic_ios

A template class for input and output streams.
The `basic_ios` template class is a base class and includes many enumerations and mechanisms necessary for input and output operations.

### 16.4.1 basic_ios Constructor

Construct an object of class `basic_ios` and assign values.

```cpp
public:
    explicit basic_ios
    (basic_streambuf<charT,traits>* sb);

protected:
    basic_ios();
```

**Remarks**

The `basic_ios` constructor creates an object to class `basic_ios` and assigns values to its member functions by calling `init()`.

### 16.4.2 Destructor

The conditions of the member functions after `init()` are shown in the following table.

```cpp
virtual ~basic_ios();
```

**Remarks**

Destroys an object of type `basic_ios`.

#### Table 16-6. Conditions after init()

<table>
<thead>
<tr>
<th>Member</th>
<th>Postcondition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdbuf()</td>
<td>sb</td>
</tr>
<tr>
<td>tie()</td>
<td>zero</td>
</tr>
<tr>
<td>rdstate()</td>
<td>goodbit if stream buffer is not a null pointer otherwise badbit.</td>
</tr>
<tr>
<td>exceptions()</td>
<td>goodbit</td>
</tr>
<tr>
<td>flags()</td>
<td>skipws</td>
</tr>
<tr>
<td>width()</td>
<td>zero</td>
</tr>
<tr>
<td>precision()</td>
<td>six</td>
</tr>
<tr>
<td>fill()</td>
<td>the space character</td>
</tr>
<tr>
<td>getloc()</td>
<td>locale::classic()</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 16-6. Conditions after init() (continued)

<table>
<thead>
<tr>
<th>Member</th>
<th>Postcondition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>iarray</td>
<td>a null pointer</td>
</tr>
<tr>
<td>parray</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

16.4.3 Basic_ios Member Functions

Member functions of the class basic_ios.

16.4.3.1 tie

To tie an ostream to the calling stream.

```cpp
basic_ostream<charT, traits>* tie() const;
basic_ostream<charT, traits>* tie(basic_ostream<charT, traits>* tiestr);
```

Remarks

Any stream can have an ostream tied to it to ensure that the ostream is flushed before any operation. The standard input and output objects cin and cout are tied to ensure that cout is flushed before any cin operation. The function tie() is overloaded. The parameterless version returns the current ostream that is tied, if any. The tie() function with an argument ties the new object to the ostream and returns a pointer, if any, from the first. The post-condition of tie() function that takes the argument tiestr is that tiestr is equal to tie();

A pointer to type ostream that is or previously tied, or zero if there was none.

Listing: Example of tie() usage:

```cpp
// The file EWL-test contains
// CodeWarrior "Software at Work"
#include <iostream>
#include <fstream>
#include <cstdlib>
char inFile[] = "EWL-test";
```
int main()
{
    using namespace std;
    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
    { cout << "file is not open"; exit(1); }
    ostream Out(inOut.rdbuf());
    if(inOut.tie())
        cout << "The streams are tied\n";
    else cout << "The streams are not tied\n";
    inOut.tie(&Out);
    inOut.rdbuf()->pubseekoff(0, ios::end);
    char str[] = "\nRegistered Trademark";
    Out << str;
    if(inOut.tie())
        cout << "The streams are tied\n";
    else cout << "The streams are not tied\n";
    inOut.close();
    return 0;
}

Result:

The streams are not tied
The streams are tied
The file EWL-test now contains
CodeWarrior "Software at Work"
Registered Trademark

16.4.3.2 rdbuf

To retrieve a pointer to the stream buffer.

basic_streambuf<charT, traits>* rdbuf() const;
basic_streambuf<charT, traits>* rdbuf(  
(basic_streambuf<charT, traits>* sb);
Remarks

To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer. The `rdbuf()` function that takes an argument has the post-condition of `sb` is equal to `rdbuf()`.

Returns a pointer to `basic_streambuf` object.

Listing: Example of `rdbuf()` usage:

```cpp
#include <iostream>
struct address {
    int number;
    char street[40];
} addbook;
int main()
{
    using namespace std;
    cout << "Enter your street number: ";
    cin >> addbook.number;
    cin.rdbuf()->pubsync(); // buffer flush
    cout << "Enter your street name: ";
    cin.get(addbook.street, 40);
    cout << "Your address is: "
         << addbook.number << " " << addbook.street;
    return 0;
}
```

Result:

Enter your street number: 1313
Enter your street name: Mockingbird Lane
Your address is: 1313 Mockingbird Lane

16.4.3.3 imbue

Stores a value representing the locale.

```cpp
locale imbue(const locale& rhs);
```
Remarks

The function `imbue()` calls `ios_base::imbue()` and `rdbuf->pubimbue()`.

Returns the current locale.

16.4.3.4 fill

To insert characters into the stream's unused spaces.

```cpp
char_type fill() const
char_type fill(char_type)
```

Remarks

Use `fill(char_type)` in output to fill blank spaces with a character. The function `fill()` is overloaded to return the current filler without altering it.

Returns the current character being used as a filler.

SeeAlso

`manipulator setfill()`

Listing: Example of fill() usage:

```cpp
#include <iostream>
int main()
{
using namespace std;
    char fill;
    cout.width(8);
    cout.fill('*');
    fill = cout.fill();
    cout << "Hi!" << "\n";
    cout << "The filler is a " << fill << endl;

    return 0;
}
```

Result:
16.4.3.5  copyfmt

Copies a basic_ios object.

    basic_ios& copyfmt(const basic_ios& rhs);

Remarks
Assigns members of *this object the corresponding objects of the rhs argument with certain exceptions. The exceptions are rdstate() is unchanged, exceptions() is altered last, and the contents or pword and iword arrays are copied not the pointers themselves. Returns the this pointer.

16.4.4  basic_ios iostate flags functions

To set flags pertaining to the state of the input and output streams.

16.4.4.1  operator bool

A bool operator.

    operator bool() const;

Remarks
Returns !fail().

16.4.4.2  operator !

A bool not operator.

    bool operator !( );
Remarks
Returns fail().

16.4.4.3 rdstate

To retrieve the state of the current formatting flags.

    iostate rdstate() const

Remarks
This member function allows you to read and check the current status of the input and output formatting flags. The returned value may be stored for use in the function ios::resetstate() to reset the flags at a later date.

Returns type iostate used in ios::resetstate()

SeeAlso
ios::resetstate()

Listing: Example of rdstate() usage:

    // The file ewl-test contains:
    // ABCDEFGHIJKLMNOPQRSTUVWXYZ
    #include <iostream>
    #include <fstream>
    #include <cstdlib>
    char * inFile = "ewl-test";
    using namespace std;
    void status(ifstream &in);
    int main()
    {
        ifstream in(inFile);
        if(!in.is_open())
        {
            cout << "could not open file for input"
                exit(1);
        }
        int count = 0;
int c;
while((c = in.get()) != EOF)
{
    // simulate a bad bit
    if(count++ == 12) in.setstate(ios::badbit);
    status(in);
}
status(in);
in.close();
return 0;
}

void status(ifstream &in)
{
    int i = in.rdstate();
    switch (i) {
    case ios::eofbit : cout << "EOF encountered \n";
        break;
    case ios::failbit : cout << "Non-Fatal I/O Error n";
        break;
    case ios::goodbit : cout << "GoodBit set \n";
        break;
    case ios::badbit : cout << "Fatal I/O Error \n";
        break;
    }
}

Result:

GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
Fatal I/O Error
16.4.4.4 clear

Clears iostate field.

```cpp
void clear
    (iostate state = goodbit) throw failure;
```

Remarks

Use `clear()` to reset the failbit, eofbit or a badbit that may have been set inadvertently when you wish to override for continuation of your processing. Post-condition of clear is the argument and is equal to rdstate().

If `rdstate()` and `exceptions()` != 0 an exception is thrown.

No value is returned.

Listing: Example of clear() usage:

```cpp
// The file ewl-test contains:
// ABCDEFGH
#include <iostream>
#include <fstream>
#include <cstdlib>
char * inFile = "ewl-test";
using namespace std;
void status(ifstream &in);
int main()
{
    ifstream in(inFile);
    if(!in.is_open())
    {
        cout << "could not open file for input";
        exit(1);
    }
    int count = 0;
    int c;
    while((c = in.get()) != EOF) {
    if(count++ == 4)
```
```cpp
// simulate a failed state
in.setstate(ios::failbit);
in.clear();
}

status(in);
}
status(in);
in.close();
return 0;
}
void status ifstream &in)
{
    // note: eof() is not needed in this example
    // if(in.eof()) cout << "EOF encountered \n"
    if(in.fail()) cout << "Non-Fatal I/O Error \n";
    if(in.good()) cout << "GoodBit set \n";
    if(in.bad()) cout << "Fatal I/O Error \n";
}

Result:

GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
Non-Fatal I/O Error

16.4.4.5 setstate

To set the state of the format flags.

    void setstate(iostate state) throw(failure);

Remarks
Calls `clear(rdstate() | state)` and may throw an exception.

There is no return value.

For an example of `setstate()` usage refer to `ios::rdstate()`

### 16.4.4.6 good

To test for the lack of error bits being set.

```cpp
bool good() const;
```

**Remarks**

Use the function `good()` to test for the lack of error bits being set.

Returns `true` if `rdstate() == 0`.

For an example of `good()` usage refer to `basic_ios::bad()`

### 16.4.4.7 eof

To test for the eofbit setting.

```cpp
bool eof() const;
```

**Remarks**

Use the `eof()` function to test for an eofbit setting in a stream being processed under some conditions. This end of file bit is not set by stream opening or closing, but only for operations that detect an end of file condition.

If `eofbit` is set in `rdstate()` `true` is returned.

**Listing: Example of `eof()` usage**

```cpp
// ewl-test is simply a one line text document
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
#include <iostream>
#include <fstream>
#include <cstdlib>
const char* TheText = "ewl-test";
```
int main()
{
    using namespace std;
    ifstream in(TheText);
    if(!in.is_open())
    {
        cout << "Couldn't open file for input";
        exit(1);
    }
    int i = 0;
    char c;
    cout.setf(ios::uppercase);
    // eofbit is not set under normal file opening
    while(!in.eof())
    {
        c = in.get();
        cout << c << " " << hex << int(c) << "\n";
        // simulate an end of file state
        if(++i == 5) in.setstate(ios::eofbit);
    }
    return 0;
}

Result:

A 41
B 42
C 43
D 44
E 45

16.4.4.8 fail

To test for stream reading failure from any cause.

    bool fail() const
Remarks

The member function fail() will test for failbit and badbit.

Returns true if failbit or badbit is set in rdstate().

Listing: Example of fail() usage

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
    using namespace std;
    char inFile[] = "ewl-test";
    ifstream in(inFile);
    if(!in.is_open())
    {cout << "Cannot open input file"; exit(1);}
    char ch = 0;
    while(!in.fail())
    {
        if(ch)cout.put(ch);
        in.get(ch);
    }

    return 0;
}
```

Result:

float 33.33 double 3.16e+10 integer 789 character C

16.4.4.9 bad

To test for fatal I/O error.
template class basic_ios

    bool bad() const

Remarks

Use the member function bad() to test if a fatal input or output error occurred which sets the badbit flag in the stream.

Returns true if badbit is set in rdstate().

SeeAlso

basic_ios::fail()

Listing: Example of bad() usage:

    // The file ewl-test contains:
    // abcdefghijklmnopqrstuvwxyz
    #include <iostream>
    #include <fstream>
    #include <cstdlib>
    char * inFile = "ewl-test";
    using namespace std;
    void status(ifstream &in);
    int main()
    {
        ifstream in(inFile);
        if(!in.is_open())
        {
            cout << "could not open file for input";
            exit(1);
        }
        int count = 0;
        int c;
        while((c = in.get()) != EOF)
        {
            // simulate a failed state
            if(count++ == 4) in.setstate(ios::failbit);
            status(in);
        }
        status(in);
    }
void status(ifstream &in)
{
    // note: eof() is not needed in this example
    // if(in.eof()) cout << "EOF encountered \n";
    if(in.fail()) cout << "Non-Fatal I/O Error \n";
    if(in.good()) cout << "GoodBit set \n";
    if(in.bad()) cout << "Fatal I/O Error \n";
}

Result:

GoodBit set
GoodBit set
GoodBit set
GoodBit set
Non-Fatal I/O Error
Non-Fatal I/O Error

16.4.4.10 exceptions

To handle basic_ios exceptions.

iostate exceptions() const;
void exceptions(iostate except);

Remarks

The function exceptions() determines what elements in rdstate() cause exceptions to be
thrown. The overloaded exceptions(iostate) calls clear(rdstate()) and leaves the argument
except equal to exceptions().

Returns a mask that determines what elements are set in rdstate().

16.5 ios_base manipulators

To provide an inline input and output formatting mechanism.
The topics in this section are:

- `fmtflags` manipulators
- `adjustfield` manipulators
- `basefield` manipulators
- `floatfield` manipulators
- Overloading Manipulators

### 16.5.1 fmtflags manipulators

To provide an inline input and output numerical formatting mechanism.

**Remarks**

Manipulators are used in the stream to alter the formatting of the stream.

A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios_base&amp; boolalpha(ios_base&amp;)</code></td>
<td>insert and extract bool type in alphabetic format</td>
</tr>
<tr>
<td><code>ios_base&amp; noboolalpha (ios_base&amp;)</code></td>
<td>unsets insert and extract bool type in alphabetic format</td>
</tr>
<tr>
<td><code>ios_base&amp; showbase(ios_base&amp; b)</code></td>
<td>set the number base to parameter b</td>
</tr>
<tr>
<td><code>ios_base&amp; noshowbase (ios_base&amp;)</code></td>
<td>remove show base</td>
</tr>
<tr>
<td><code>ios_base&amp; showpoint(ios_base&amp;)</code></td>
<td>show decimal point</td>
</tr>
<tr>
<td><code>ios_base&amp; noshowpoint(ios_base&amp;)</code></td>
<td>do not show decimal point</td>
</tr>
<tr>
<td><code>ios_base&amp; showpos(ios_base&amp;)</code></td>
<td>show the positive sign</td>
</tr>
<tr>
<td><code>ios_base&amp; noshowpos(ios_base&amp;)</code></td>
<td>do not show positive sign</td>
</tr>
<tr>
<td><code>ios_base&amp; skipws(ios_base&amp;)</code></td>
<td>input only skip white spaces</td>
</tr>
<tr>
<td><code>ios_base&amp; noskipws(ios_base&amp;)</code></td>
<td>input only no skip white spaces</td>
</tr>
<tr>
<td><code>ios_base&amp; uppercase(ios_base&amp;)</code></td>
<td>show scientific in uppercase</td>
</tr>
<tr>
<td><code>ios_base&amp; nouppercase (ios_base&amp;)</code></td>
<td>do not show scientific in uppercase</td>
</tr>
<tr>
<td><code>ios_base&amp; unitbuf (ios_base::unitbuf)</code></td>
<td>set the unitbuf flag</td>
</tr>
<tr>
<td><code>ios_base&amp; nounitbuf (ios_base::unitbuf)</code></td>
<td>unset the unitbuf flag</td>
</tr>
</tbody>
</table>

### 16.5.2 adjustfield manipulators

To provide an inline input and output orientation formatting mechanism.

**Remarks**
Manipulators are used in the stream to alter the formatting of the stream.

A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

### Table 16-8. Adjustfield manipulators

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios_base&amp; internal(ios_base&amp;)</code></td>
<td>fill between indicator and value</td>
</tr>
<tr>
<td><code>ios_base&amp; left(ios_base&amp;)</code></td>
<td>left justify in a field</td>
</tr>
<tr>
<td><code>ios_base&amp; right(ios_base&amp;)</code></td>
<td>right justify in a field</td>
</tr>
</tbody>
</table>

#### 16.5.3 `basefield manipulators`

To provide an inline input and output numerical formatting mechanism.

**Remarks**

Manipulators are used in the stream to alter the formatting of the stream.

A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

### Table 16-9. Basefield manipulators

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios_base&amp; dec(ios_base&amp;)</code></td>
<td>format output data as a decimal</td>
</tr>
<tr>
<td><code>ios_base&amp; oct(ios_base&amp;)</code></td>
<td>format output data as octal</td>
</tr>
<tr>
<td><code>ios_base&amp; hex(ios_base&amp;)</code></td>
<td>format output data as hexadecimal</td>
</tr>
</tbody>
</table>

#### 16.5.4 `floatfield manipulators`

To provide an inline input and output numerical formatting mechanism.

**Remarks**

Manipulators are used in the stream to alter the formatting of the stream.

A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

### Table 16-10. Floatfield manipulators

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios_base&amp; fixed(ios_base&amp;)</code></td>
<td>format in fixed point notation</td>
</tr>
<tr>
<td><code>ios_base&amp; scientific(ios_base&amp;)</code></td>
<td>use scientific notation</td>
</tr>
</tbody>
</table>
Listing: Example of manipulator usage:

```cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    long number = 64;
    cout << "Original Number is "
         << number << "\n\n";
    cout << showbase;
    cout << setw(30) << "Hexadecimal :" << hex << setw(10) << right
         << number <<'\n';
    cout << setw(30) << "Octal :" << oct
         << setw(10) << left
         << number <<'\n';
    cout << setw(30) << "Decimal :" << dec
         << setw(10) << right
         << number << endl;
    return 0;
}
```

Result:

Original Number is 64
Hexadecimal : 0x40
Octal :0100
Decimal : 64

16.5.5 Overloading Manipulators

To provide an inline formatting mechanism. The basic template for parameterless manipulators is shown in the listing below.

Listing: Basic parameterless manipulator

```cpp
ostream &manip-name(ostream &stream)
{
    // coding
    return stream;
}
```
Remarks

Use overloaded manipulators to provide specific and unique formatting methods relative to one class.

A reference to ostream. (Usually the this pointer.)

See Also

<iomanip> for manipulators with parameters

Listing: Example of overloaded manipulator usage:

```cpp
#include <iostream>
using namespace std;
ostream &rJus(ostream &stream);

int main()
{
    cout << "align right " << rJus << "for column";
    return 0;
}

ostream &rJus(ostream &stream)
{
    stream.width(30);
    stream.setf(ios::right);
    return stream;
}

Result:

align right for column
Chapter 17
Stream Buffers

The header `<streambuf>` defines types that control input and output to character sequences.

Listing: Header `<streambuf>`

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT> >
        class basic_streambuf;
    typedef basic_streambuf<char> streambuf;
    typedef basic_streambuf<wchar_t> wstreambuf;
}
```

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Stream buffer requirements
- Class basic_streambuf

17.1 Stream buffer requirements

This section explains the constraints Stream buffers can impose.

The constraints include:

- The input sequence can be not readable
- The output sequence can be not writable
- The sequences can be associated with other presentations such as external files
- The sequences can support operations to or from associated sequences.
- The sequences can impose limitations on how the program can read and write characters to and from a sequence or alter the stream position.
There are three pointers that control the operations performed on a sequence or associated sequences. These are used for read, writes and stream position alteration. If not null all pointers point to the same \texttt{charT} array object.

- The beginning pointer or lowest element in an array. - (\texttt{beg})
- The next pointer of next element addressed for read or write. - (\texttt{next})
- The end pointer of first element addressed beyond the end of the array. - (\texttt{end})

\section{17.2 Class \texttt{basic_streambuf}}

The template class \texttt{basic_streambuf} is an abstract class for deriving various stream buffers whose objects control input and output sequences.

The type \texttt{streambuf} is an instantiation of \texttt{char} type. the type \texttt{wstreambuf} is an instantiation of \texttt{wchar_t} type.

The prototype is listed below. Additional topics in this section are:

- \texttt{basic_streambuf Constructor}
- \texttt{basic_streambuf Public Member Functions}
- Locales
- Buffer Management and Positioning
- Get Area
- Putback
- Put Area
- \texttt{basic_streambuf Protected Member Functions}
- Get Area Access
- Put Area Access
- \texttt{basic_streambuf Virtual Functions}
- Buffer Management and Positioning
- Get Area
- Putback
- Put Area

\subsection{17.2.1 \texttt{basic_streambuf Constructor}}

The default constructor constructs an object of type \texttt{basic_streambuf}. 

protected:
    basic_streambuf();

Remarks
The constructor sets all pointer member objects to null pointers and calls getloc() to copy the global locale at the time of construction.

17.2.1.1 Destructor

virtual ~basic_streambuf();

Remarks
Removes the object from memory.

17.2.2 basic_streambuf Public Member Functions

The public member functions allow access to member functions from derived classes.

17.2.2.1 Locales

Locales are used for encapsulation and manipulation of information to a particular locale.

17.2.2.2 basic_streambuf::pubimbue

To set the locale.

    locale pubimbue(const locale &loc);

Remarks
The function pubimbue calls imbue(loc).

Returns the previous value of getloc().
17.2.3  basic_streambuf::getloc

To get the locale.

locale getloc() const;

Remarks
If pubimbue has already been called, it returns the last value of loc supplied; otherwise the current one. If pubimbue has been called but has not returned a value from imbue, it then returns the previous value.

17.2.3  Buffer Management and Positioning
Functions used to manipulate the buffer and the input and output positioning pointers.

17.2.3.1  basic_streambuf::pubsetbuf

To set an allocation after construction.

basic_streambuf<char_type, traits> *pubsetbuf
        (char_type* s, streamsize n);

Remarks
The first argument is used in another function by a filebuf derived class. See setbuf(). The second argument is used to set the size of a dynamic allocated buffer.

Returns a pointer to basic_streambuf<char_type, traits> via setbuf(s, n).

Listing: Example of basic_streambuf::pubsetbuf() usage:

#include <iostream>
#include <sstream>
const int size = 100;
char temp[size] = "\0";
int main()
{

using namespace std;

stringbuf strbuf;
strbuf.pubsetbuf('\0', size);
strbuf.sputn("CodeWarrior", 50);
strbuf.sgetn(temp, 50);
     cout << temp;
     return 0;
}

Result:

CodeWarrior

17.2.3.2 basic_streambuf::pubseekoff

Determines the position of the get pointer.

    pos_type pubseekoff
       (off_type off,
        ios_base::seekdir way, ios_base::openmode
        which = ios_base::in | ios_base::out);

Remarks

The member function pubseekoff() is used to find the difference in bytes of the get pointer from a known position (such as the beginning or end of a stream). The function pubseekoff() returns a type pos_type which holds all the necessary information.

Returns a pos_type via seekoff(off, way, which)

See Also

pubseekpos()

Listing: Example of basic_streambuf::pubseekoff() usage:

    // The ewl-test file contains originally
    // CodeWarrior "Software at Work"

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <stdio.h>
char inFile[] = "ewl-test";
```cpp
int main() {
    using namespace std;
    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open()) {
        cout << "Could not open file"; exit(1);
    }

    ostream Out(inOut.rdbuf());
    char str[] = "\nRegistered Trademark";
    inOut.rdbuf()->pubseekoff(0, ios::end);
    Out << str;
    inOut.close();
    return 0;
}
```

Result:

The File now reads:
CodeWarrior "Software at Work"
Registered Trademark

17.2.3.3 basic_streambuf::pubseekpos

Determine and move to a desired offset.

```cpp
pos_type pubseekpos
(pos_type sp,
    ios_base::openmode which = ios::in |ios::out);
```

Remarks

The function `pubseekpos()` is used to move to a desired offset using a type `pos_type`, which holds all necessary information.

Returns a `pos_type` via `seekpos(sb, which)`

See Also

`pubseekoff()`, `seekoff()`

Listing: Example of streambuf::pubseekpos() usage:

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

int main() {
    using namespace std;
    ifstream in("ewl-test");
    if(!in.is_open()) {
        cout << "could not open file"; exit(1);
    }

    streampos spEnd(0), spStart(0), aCheck(0);
    spEnd = spStart = 5;
```
aCheck = in.rdbuf()->pubseekpos(spStart,ios::in);

cout << "The offset at the start of the reading" << " in bytes is " << static_cast<streamoff>(aCheck) << endl;

char ch;
while(spEnd != spStart+10)
{
   in.get(ch);
   cout << ch; 
   spEnd = in.rdbuf()->pubseekoff(0, ios::cur);
}

aCheck = in.rdbuf()->pubseekoff(0,ios::cur);
cout << "The final position's offset in bytes now is " << static_cast<streamoff>(aCheck) << endl;

in.close();
return 0;
}

Result:

The offset for the start of the reading in bytes is 5
FGHIJKL
The final position's offset in bytes now is 15

17.2.3.4 basic_streambuf::pubsync

To synchronize the streambuf object with its input/output.

    int pubsync();

Remarks

The function pubsync() will attempt to synchronize the streambuf input and output.

Returns zero if successful or EOF if not via sync().

Listing: Example of streambuf::pubsync() usage:

#include <iostream>
struct address {
  int number;
  char street[40];
}addbook;
int main()
{
using namespace std;
  cout << "Enter your street number: ";
cin >> addbook.number;

    cin.rdbuf()->pubsync(); // buffer flush

    cout << "Enter your street name: ";

    cin.get(addbook.street, 40);

    cout << "Your address is: "

        << addbook.number << " " << addbook.street;

    return 0;
}

Result:

Enter your street number: 2201
Enter your street name: Donley Drive
Your address is: 2201 Donley Drive

17.2.4  Get Area

Public functions for retrieving input from a buffer.

17.2.4.1 basic_streambuf::in_avail

To test for availability of input stream.

    streamsize in_avail();

Remarks

If a read is permitted returns size of stream as a type streamsize.

17.2.4.2 basic_streambuf::snextc

To retrieve the next character in a stream.

    int_type snextc();

Remarks
The function \texttt{snextc()} calls \texttt{sbumpc()} to extract the next character in a stream. After the operation, the get pointer references the character following the last character extracted.

If \texttt{sbumpc} returns \texttt{traits::eof}, otherwise returns \texttt{sgetc()}.

**Listing: Example of streambuf::snextc() usage:**

```cpp
#include <iostream>
#include <sstream>
const int size = 100;
int main()
{
    using namespace std;
    stringbuf strbuf;
    strbuf.pubsetbuf('\0', size);
    strbuf.sputn("ABCDE", 50);
    char ch;
    // look ahead at the next character
    ch = strbuf.snextc();
    cout << ch;
    // get pointer was not returned after peeking
    ch = strbuf.snextc();
    cout << ch;
    return 0;
}
```

Result:

```
BC
```

### 17.2.4.3 basic\_streambuf::sbumpc

To move the get pointer.

```cpp
int_type sbumpc();
```

**Remarks**

The function \texttt{sbumpc()} moves the get pointer one element when called.
Return

The value of the character at the get pointer. It returns uflow() if it fails to move the pointer.

See Also

sgetc()

Listing: Example of streambuf::sbumpc() usage:

```cpp
#include <iostream>
#include <sstream>

const int size = 100;
std::string buf = "CodeWarrior --Software at Work--";

int main()
{
    using namespace std;
    stringbuf strbuf(buf);
    int ch;
    for (int i = 0; i < 23; i++)
    {
        ch = strbuf.sgetc();
        strbuf.sbumpc();
        cout.put(ch);
    }
    cout << endl;
    cout << strbuf.str() << endl;
    return 0;
}
```

Result:

CodeWarrior
CodeWarrior --Software at Work--

17.2.4.4 basic_streambuf::sgetc

To extract a character from the stream.

```cpp
int_type sgetc();
```

Remarks

The function sgetc() extracts a single character, without moving the get pointer.

A int_type type at the get pointer if available, otherwise returns underflow().

For an example of streambuf::sgetc() usage refer to streambuf::sbumpc()
17.2.4.5 basic_streambuf::sgetn

To extract a series of characters from the stream.

```c
streamsize sgetn(char_type *s, streamsize n);
```

Remarks

The public member function `sgetn()` is used to extract a series of characters from the stream buffer. After the operation, the get pointer references the character following the last character extracted.

Returns a `streamsize` type as returned from the function `xsgetn(s,n)`.

For an example of streambuf::sgetn() usage refer to `pubsetbuf()`

17.2.5 Putback

Public functions to return a value to a stream.

17.2.5.1 basic_streambuf::sputbackc

To put a character back into the stream.

```c
int_type sputbackc(char_type c);
```

Remarks

The function `sputbackc()` will replace a character extracted from the stream with another character. The results are not assured if the putback is not immediately done or a different character is used.

If successful, returns a pointer to the get pointer as an `int_type` otherwise returns `pbackfail(c)`.

Listing: Example of streambuf::sputbackc() usage:
```cpp
#include <iostream>
#include <sstream>

std::string buffer = "ABCDEF";

int main()
{
    using namespace std;

    stringbuf strbuf(buffer);
    char ch;

    ch = strbuf.sgetc(); // extract first character
    cout << ch;        // show it
    // get the next character
    ch = strbuf.snextc();
    // if second char is B replace first char with x
    if(ch == 'B') strbuf.sputbackc('x');
    // read the first character now x
    cout << (char)strbuf.sgetc();
    strbuf.sbumpc();      // increment get pointer
    // read second character
    cout << (char)strbuf.sgetc();
    strbuf.sbumpc();      // increment get pointer
    // read third character
    cout << (char)strbuf.sgetc();

    // show the new stream after alteration
    strbuf.pubseekoff(0, ios::beg);
    cout << endl;

    cout << (char)strbuf.sgetc();

    while( (ch = strbuf.snextc()) != EOF)
    {
        cout << ch;
    }

    return 0;
}
```
Result:

AxB

xBCDEF

17.2.5.2 basic_streambuf::sungetc

To restore a character extracted.

```cpp
int_type sungetc();
```

**Remarks**

The function `sungetc()` restores the previously extracted character. After the operation, the get pointer references the last character extracted.

If successful, returns a pointer to the get pointer as an int_type otherwise returns `pbackfail(c)`.

For an example of streambuf::sungetc() usage refer to streambuf::sputbackc()

17.2.6 Put Area

Public functions for inputting characters into a buffer.

17.2.6.1 basic_streambuf::sputc

To insert a character in the stream.

```cpp
int_type sputc(char_type c);
```

**Remarks**
The function \texttt{sputc()} inserts a character into the stream. After the operation, the get pointer references the character following the last character inserted.

If successful, returns \texttt{c} as an \texttt{int\_type} otherwise returns \texttt{overflow(c)}.

\textbf{Listing: Example of \texttt{streambuf::sputc()} usage:}

```cpp
#include <iostream>
#include <sstream>

int main()
{
    using namespace std;
    stringbuf strbuf;
    strbuf.sputc('A');
    char ch;
    ch = strbuf.sgetc();
    cout << ch;
    return 0;
}
```

\textbf{Result:}

\texttt{A}

\textbf{17.2.6.2 basic\_streambuf::sputn}

To insert a series of characters into a stream.

```cpp
int\_type sputn(char\_type *s, streamsize n);
```

\textbf{Remarks}

The function \texttt{sputn()} inserts a series of characters into a stream. After the operation, the get pointer references the character following the last character inserted.

Returns a \texttt{streamsize} type returned from a call to \texttt{xputn(s,n)}.

\textbf{17.2.6.3 basic\_streambuf Protected Member Functions}
Protected member functions that are used for stream buffer manipulations by the
\texttt{basic\_streambuf} class and derived classes from it.

\section*{17.2.7 Get Area Access}

Member functions for extracting information from a stream.

\subsection*{17.2.7.1 \texttt{basic\_streambuf\::eback}}

Retrieve the beginning pointer for stream input.

\begin{verbatim}
char\_type* eback() const;
\end{verbatim}

\textbf{Remarks}

Returns the beginning pointer.

\subsection*{17.2.7.2 \texttt{basic\_streambuf\::gptr}}

Retrieve the next pointer for stream input.

\begin{verbatim}
char\_type* gptr() const;
\end{verbatim}

\textbf{Remarks}

Returns the next pointer.

\subsection*{17.2.7.3 \texttt{basic\_streambuf\::egptr}}

Retrieve the end pointer for stream input.

\begin{verbatim}
char\_type* egptr() const;
\end{verbatim}

\textbf{Remarks}

Returns the end pointer.
17.2.7.4  basic_streambuf::gbump

Advances the next pointer for stream input.

    void gbump(int n);

Remarks
The function gbump() advances the input pointer by the value of the int n argument.

17.2.7.5  basic_streambuf::setg

To set the beginning, next and end pointers.

    void setg
        (char_type *gbeg,
         char_type *gnext,
         char_type *gend);

Remarks
After the call to setg() the gbeg pointer equals eback(), the gnext pointer equals gptr(),
and the gend pointer equals egptr().

17.2.8  Put Area Access

Protected member functions for stream output sequences.

17.2.8.1  basic_streambuf::pbase

To retrieve the beginning pointer for stream output.

    char_type* pbase() const;

Remarks
Returns the beginning pointer.

17.2.8.2  basic_streambuf::pptr

To retrieve the next pointer for stream output.

    char_type* pptr() const;

Remarks
Returns the next pointer.

17.2.8.3  basic_streambuf::eptr

To retrieve the end pointer for stream output.

    char_type* eptr() const;

Remarks
Returns the end pointer.

17.2.8.4  basic_streambuf::pbump

To advance the next pointer for stream output.

    void pbump(int n);

Remarks
The function pbump() advances the next pointer by the value of the int argument n.

17.2.8.5  basic_streambuf::setp

To set the values for the beginning, next and end pointers.
void setp
   (char_type* pbeg,
    char_type* pend);

Remarks
After the call to setp(), pbeg equals pbase(), pbeg equals pptr() and pend equals epptr().

17.2.9 basic_streambuf Virtual Functions

The virtual functions in basic_streambuf class are to be overloaded in any derived class.

17.2.9.1 Locales

To get and set the stream locale. These functions should be overridden in derived classes.

17.2.9.2 basic_streambuf::imbue

To change any translations base on locale.

   virtual void imbue(const locale &loc);

Remarks
The imbue() function allows the derived class to be informed in changes of locale and to cache results of calls to locale functions.

17.2.10 Buffer Management and Positioning

Virtual functions for positioning and manipulating the stream buffer. These functions should be overridden in derived classes.
17.2.10.1 basic_streambuf::setbuf

To set a buffer for stream input and output sequences.

virtual basic_streambuf<char_type, traits> *setbuf
  (char_type* s, streamsize n);

Remarks
The function setbuf() is overridden in basic_stringbuf and basic_filebuf classes.
Returns the this pointer.

17.2.10.2 basic_streambuf::seekoff

To return an offset of the current pointer in an input or output stream.

virtual pos_type seekoff
  (off_type off,
   ios_base::seekdir way,
   ios_base::openmode which = ios::in |ios::out);

Remarks
The function seekoff() is overridden in basic_stringbuf and basic_filebuf classes.
Returns a pos_type value, which is an invalid stream position.

17.2.10.3 basic_streambuf::seekpos

To alter an input or output stream position.

virtual pos_type seekpos
  (pos_type sp,
   ios_base::openmode which = ios::in |ios::out);

Remarks
The function seekpos() is overridden in basic_stringbuf and basic_filebuf classes.
Returns a pos_type value, which is an invalid stream position.
17.2.10.4 basic_streambuf::sync

To synchronize the controlled sequences in arrays.

virtual int sync();

Remarks

If pbase() is non null the characters between pbase() and pptr() are written to the control sequence. The function setbuf() overrides the basic_filebuf class.

Returns zero if successful and -1 if failure occurs.

17.2.11 Get Area

Virtual functions for extracting information from an input stream buffer. These functions should be overridden in derived classes.

17.2.11.1 basic_streambuf::showmanyc

Shows how many characters in an input stream

virtual int showmanyc();

Remarks

The function returns zero for the default behavior. Derived classes may return a negative one or a non-negative value. A positive value estimates the number of characters available in the sequence. If a positive value is returned, then successive calls to underflow() will not return traits::eof() until at least that number of characters have been extracted from the stream. If showmanyc() returns -1, then calls to underflow() or uflow() will fail.

Note that underflow or uflow might fail by throwing an exception prematurely. The intention is that the calls will not return eof() and will return immediately.
17.2.11.2  basic_streambuf::xsgetn

To read a number of characters from an input stream buffer.

    virtual streamsize xsgetn
    (char_type *s, streamsize n);

Remarks
The characters are read by repeated calls to sbumpc() until either n characters have been assigned or EOF is encountered.

Returns the number of characters read.

17.2.11.3  basic_streambuf::underflow

To show an underflow condition and not increment the get pointer.

    virtual int_type underflow();

Remarks
The function underflow() is called when a character is not available for sgetc(). There are many constraints for underflow().

The pending sequence of characters is a concatenation of end pointer minus the get pointer plus some sequence of characters to be read from input.

The result character if the sequence is not empty, the first character in the sequence or the next character in the sequence.

The backup sequence if the beginning pointer is null, the sequence is empty, otherwise the sequence is the get pointer minus the beginning pointer.

Returns the first character of the pending sequence and does not increment the get pointer. If the position is null returns traits::eof() to indicate failure.

17.2.11.4  basic_streambuf::uflow

To show a underflow condition for a single character and increment the get pointer.
virtual int_type uflow();

Remarks
The function uflow() is called when a character is not available for sbumpc().

The constraints are the same as underflow(), with the exceptions that the resultant character is transferred from the pending sequence to the back up sequence and the pending sequence may not be empty.

Calls underflow() and if traits::eof is not returned returns the integer value of the get pointer and increments the next pointer for input.

17.2.12 Putback
Virtual functions for replacing data to a stream. These functions should be overridden in derived classes.

17.2.12.1 basic_streambuf::pbackfail
To show a failure in a put back operation.

    virtual int_type pbackfail
    (int_type c = traits::eof());

Remarks
The resulting conditions are the same as the function underflow().

The function pbackfail() is only called when a put back operation really has failed and returns traits::eof. If success occurs the return is undefined.

17.2.13 Put Area
Virtual function for inserting data into an output stream buffer.
These functions should be overridden in derived classes.
17.2.13.1 basic_streambuf::xsputn

Write a number of characters to an output buffer.

```cpp
virtual streamsize xsputn
    (const char_type *s, streamsize n);
```

Remarks

The function `xsputn()` writes to the output character by using repeated calls to `sputc(c)`. Write stops when `n` characters have been written or `EOF` is encountered.

Returns the number of characters written in a type `streamsize`.

17.2.13.2 basic_streambuf::overflow

Consumes the pending characters of an output sequence.

```cpp
virtual int_type overflow
    (int_type c = traits::eof());
```

Remarks

The pending sequence is defined as the concatenation of the put pointer minus the beginning pointer plus either the sequence of characters or an empty sequence, unless the beginning pointer is null in which case the pending sequence is an empty sequence.

This function is called by `sputc()` and `sputn()` when the buffer is not large enough to hold the output sequence.

Overriding this function requires that:

When overridden by a derived class how characters are consumed must be specified.

After the overflow either the beginning pointer must be null or the beginning and put pointer must both be set to the same non-null value.

The function may fail if appending characters to an output stream fails or failure to set the previous requirement occurs.

The function returns `traits::eof()` for failure or some unspecified result to indicate success.
Chapter 18
Formatting and Manipulators

This chapter discusses formatting and manipulators in the input/output library.

There are three headers- `<istream>`, `<ostream>`, and `<iomanip>`-that contain stream formatting and manipulator routines and implementations.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Headers
- Input Streams
- Output streams
- Standard manipulators

18.1 Headers

This section lists the header for istream, ostream, and iomanip.

- Header `<istream>` for input streams
- Header `<ostream>` for output streams
- Header `<iomanip>` for input and output manipulation

18.2 Input Streams

The header `<istream>` controls input from a stream buffer.

The topics in this section are:

- Template class basic_istream
- Class basic_istream::sentry
- Formatted input functions
18.2.1 Template class basic_istream

A class that defines several functions for stream input mechanisms from a controlled stream buffer.

The basic_istream class is derived from the basic_ios class and provides many functions for input operations.

18.2.1.1 basic_istream Constructors

Creates a basic_istream object.

```cpp
explicit basic_istream
    (basic_streambuf<charT, traits>* sb);
```

Remarks

The basic_istream constructor is overloaded. It can be created as a base class with no arguments. It may be a simple input class initialized to a previous object's stream buffer.

18.2.1.2 Destructor

Destroy the basic_istream object.

```cpp
virtual ~basic_istream();
```

Remarks

The basic_istream destructor removes from memory the basic_istream object.

Listing: Example of basic_istream() usage:

```cpp
// ewl-test file contains
// Ask the teacher anything you want to know
#include <iostream>
```
```cpp
#include <fstream>
#include <cstdlib>

int main()
{
    using namespace std;

    ofstream out("ewl-test", ios::out | ios::in);
    if(!out.is_open())
    {
        cout << "file did not open"; exit(1);
    }

    istream inOut(out.rdbuf());

    char c;
    while(inOut.get(c)) cout.put(c);
    return 0;
}
```

Result:

Ask the teacher anything you want to know

### 18.2.2 Class `basic_istream::sentry`

A class for exception safe prefix and suffix operations.

#### 18.2.2.1 Class `basic_istream::sentry` Constructor

Prepare for formatted or unformatted input

```cpp
explicit sentry
    (basic_istream<charT, traits>& is, bool noskipws = false);
```

Remarks
If after the operation \texttt{is.good()} is true \texttt{ok_} equals \texttt{true} otherwise \texttt{ok_} equals \texttt{false}. The constructor may call \texttt{setstate(failbit)} which may throw an exception.

18.2.2.2 Destructor
Destroys a sentry object.

\begin{verbatim}
-\texttt{sentry}();
\end{verbatim}

Remarks
The destructor has no effects.

18.2.2.3 \texttt{sentry::Operator bool}
To return the value of the data member \texttt{ok_}.

\begin{verbatim}
\texttt{operator bool}();
\end{verbatim}

Remarks
Operator bool returns the value of \texttt{ok_}

18.2.3 Formatted input functions
Formatted functions provide mechanisms for input operations of specific types.

18.2.3.1 Common requirements
Each formatted input function begins by calling \texttt{ipfx()} and if the scan fails for any reason, then calls \texttt{setstate(failbit)}. The behavior of the scan functions are "as if" it was \texttt{fscanf}().

18.2.3.2 Arithmetic Extractors Operator >>
Extractors that provide formatted arithmetic input operations. Each signature extracts the specified type and stores it in \( n \).

\[
\begin{align*}
\text{basic\_istream\& operator } & \gg (\text{bool } & n); \\
\text{basic\_istream\& operator } & \gg (\text{short } & n); \\
\text{basic\_istream\& operator } & \gg (\text{unsigned short } & n); \\
\text{basic\_istream\& operator } & \gg (\text{int } & n); \\
\text{basic\_istream\& operator } & \gg (\text{unsigned int } & n); \\
\text{basic\_istream\& operator } & \gg (\text{long } & n); \\
\text{basic\_istream\& operator } & \gg (\text{unsigned long } & n); \\
\text{basic\_istream\& operator } & \gg (\text{float } & f); \\
\text{basic\_istream\& operator } & \gg (\text{double } & f); \\
\text{basic\_istream\& operator } & \gg (\text{long double } & f);
\end{align*}
\]

**Remarks**

The Arithmetic extractors extract a specific type from the input stream and store it in the address provided.

**Table 18-1. States and stdio equivalents**

<table>
<thead>
<tr>
<th>state</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{flags() } &amp; \text{basefield}) = \text{oct})</td>
<td>(%o)</td>
</tr>
<tr>
<td>((\text{flags() } &amp; \text{basefield}) = \text{hex})</td>
<td>(%x)</td>
</tr>
<tr>
<td>((\text{flags() } &amp; \text{basefield}) \neq 0)</td>
<td>(%x)</td>
</tr>
<tr>
<td>((\text{flags() } &amp; \text{basefield}) = 0)</td>
<td>(%i)</td>
</tr>
<tr>
<td>Otherwise</td>
<td></td>
</tr>
<tr>
<td>signed integral type</td>
<td>(%d)</td>
</tr>
<tr>
<td>unsigned integral type</td>
<td>(%u)</td>
</tr>
</tbody>
</table>

### 18.2.3.3 basic\_istream extractor operator \(\gg\)

Extracts characters or sequences of characters and converts if necessary to numerical data.

\[
\begin{align*}
\text{basic\_istream\& operator } & \gg \\
\text{(basic\_istream\& (*pf)}
\end{align*}
\]

**Returns** \(\text{pf(*this)}\).

\[
\begin{align*}
\text{basic\_istream\& operator } & \gg \\
\text{(basic\_ios\& (*pf)(basic\_ios\&))}
\end{align*}
\]

**Calls** \(\text{pf(*this)}\) then returns \(\text{*this}\).
basic_istream<charT, traits>& operator >>(char_type *s);

Extracts a char array and stores it in \( s \) if possible otherwise calls setstate(failbit). If width() is set greater than zero width()-1, elements are extracted; otherwise up to size of s-l elements are extracted. Scan stops with a whitespace "as if" in fscanf().

basic_istream<charT, traits>& operator >>(char_type& c);

Extracts a single character and stores it in \( c \) if possible, otherwise calls setstate(failbit).

basic_istream<charT, traits>& operator >>(void*& p);

Converts a pointer to void and stores it in \( p \).

basic_istream<charT, traits>& operator >>(basic_streambuf<char_type, traits>* sb);

Extracts a basic_streambuf type and stores it in \( sb \) if possible, otherwise calls setstate(failbit).

Remarks

The various overloaded extractors are used to obtain formatted input dependent upon the type of argument. Since they return a reference to the calling stream they may be chained in a series of extractions. The overloaded extractors work "as if" like fscanf() in standard C and read until a white space character or EOF is encountered.

The white space character is not extracted and is not discarded, but simply ignored. Be careful when mixing unformatted input operations with the formatted extractor operators, such as when using console input.

The this pointer is returned.

See Also

basic_ostream::operator

Listing: Example of basic_istream:: extractor usage:

```cpp
// The ewl-test input file contains
// float 33.33 double 3.16e+10 Integer 789 character C

#include <iostream>
#include <fstream>
#include <cstdlib>

char ioFile[81] = "ewl-test";

int main()
{
    ifstream in(ioFile);
    if(!in.is_open())
    {
        cout << "cannot open file for input"; exit(1);
    }
char type[20];
double d;
int i;
char ch;

in >> type >> d;
cout << type << " " << d << endl;
in >> type >> d;
cout << type << " " << d << endl;
in >> type >> i;
cout << type << " " << i << endl;
in >> type >> ch;
cout << type << " " << ch << endl;
cout << "\nEnter an integer: ";

in >> i;
cout << "Enter a word: ";

in >> type;
cout << "Enter a character \ "
    << " then a space then a double: ";

in >> ch >> d;
cout << i << " " << type << " 
    << ch << " " << d << endl;
in.close();
return 0;
}

Result:

float 33.33
double 3.16e+10
Integer 789
character C
Enter an integer: 123 <enter>
Enter a word: CodeWarrior <enter>
Enter a character then a space then a double: a 12.34 <enter>
123 CodeWarrior a 12.34

18.2.3.4 Overloading Extractors

To provide custom formatted data retrieval.

extractor prototype
Basic_istream &operator>>(basic_istream &s, const imanip<T>&)
{
    // procedures
    return s;
}
Remarks

You may overload the extractor operator to tailor the specific needs of a particular class.

The this pointer is returned.

Listing: Example of basic_istream overloaded extractor usage:

```cpp
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

class phonebook {
    friend std::ostream &operator<<(std::ostream &stream, phonebook o);
    friend std::istream &operator>>(std::istream &stream, phonebook &o);

private:
    char name[80];
    int areacode;
    int exchange;
    int num;

public:
    void putname() {std::cout << num;}
    phonebook() {};    // default constructor
    phonebook(char *n, int a, int p, int nm) {
    {std::strcpy(name, n); areacode = a;
        exchange = p; num = nm;}
    }

int main()
{
    using namespace std;
    phonebook a;
    cin >> a;
    cout << a;
    return 0;
}

std::ostream &operator<<(std::ostream &stream, phonebook o)
```
{ 
using namespace std;
    stream << o.name << " ";
    stream << "(" << o.areacode << " ) " ;
    stream << o.exchange << "-" ;
    cout << setfill( '0' ) << setw( 4 ) << o.num << "\n" ;
    return stream;
} 
std::istream &operator>>(std::istream &stream, phonebook &o) 
{ 
using namespace std;
    char buf[5];
    cout << "Enter the name: " ;
    stream >> o.name;
    cout << "Enter the area code: " ;
    stream >> o.areacode;
    cout << "Enter exchange: " ;
    stream >> o.exchange;
    cout << "Enter number: " ;
    stream >> buf;
    o.num = atoi(buf);
    cout << "\n" ;
    return stream;
}
Result:

Enter the name: CodeWarrior
Enter the area code: 512
Enter exchange: 996
Enter number: 5300
CodeWarrior (512) 996-5300

18.2.4 Unformatted input functions

The various unformatted input functions all begin by constructing an object of type basic_istream::sentry and ends by destroying the sentry object.
NOTE
Older versions of the library may begin by calling ipfx() and end by calling isfx() and returning the value specified.

18.2.4.1 basic_istream::gcount

To obtain the number of bytes read.

    streamsize gcount() const;

Remarks

Use the function gcount() to obtain the number of bytes read by the last unformatted input function called by that object.

Returns an int type count of the bytes read.

Listing: Example of basic_istream::gcount() usage:

```cpp
#include <iostream>
#include <fstream>
const SIZE = 4;
struct stArray {
    int index;
    double dNum;
};
int main()
{
    using namespace std;
    ofstream fOut("test");
    if(!fOut.is_open())
    {
        cout << "can't open out file"; return 1;
    }
    stArray arr;
    short i;
    for(i = 1; i < SIZE+1; i++)
    {
        arr.index = i;
        arr.dNum = i * 3.14;
        fOut.write((char *) &arr, sizeof(stArray));
    }
```
fOut.close();
stArray aIn[SIZE];
ifstream fIn("test");
if(!fIn.is_open())
    {cout << "can't open in file"; return 2;}
long count =0;
for(i = 0; i < SIZE; i++)
{
    fIn.read((char *) &aIn[i], sizeof(stArray));
count+=fIn.gcount();
}
cout << count << " bytes read " << endl;
cout << "The size of the structure is "
    << sizeof(stArray) << endl;
for(i = 0; i < SIZE; i++)
cout << aIn[i].index << " " << aIn[i].dNum
    << endl;
fIn.close();
return 0;
}

Result:

48 bytes read
The size of the structure is 12
1 3.14
2 6.28
3 9.42
4 12.56

18.2.4.2  basic_istream::get

Overloaded functions to retrieve a char or a char sequence from an input stream.

    int_type get();

Extracts a character if available and returns that value, otherwise calls setstate(failbit) and returns eof().
basic_istream<charT, traits>& get(char_type& c);

Extracts a character and assigns it to \( c \) if possible, else calls setstate(failbit).

basic_istream<charT, traits>& get(char_type* s, streamsize n, char_type delim = traits::newline());

18.2.4.2.1 Remarks

Extracts characters and stores them in a char array at an address pointed to by \( s \), until:

- a limit (the second argument minus one) or the number of characters to be stored is reached
- a delimiter (the default value is the newline character) is met. In which case, the delimiter is not extracted.

If end_of_file is encountered, setstate(eofbit) is called.

If no characters are extracted setstate(failbit) is called. In any case, it stores a null character in the next available location of array \( s \).

basic_istream<charT, traits>& get (basic_streambuf<char_type, traits>& sb, char_type delim = traits::newline());

Extracts characters and assigns them to the basic_streambuf object \( sb \) if possible or else it calls setstate(failbit). Extraction stops if:

- an insertion fails
- end-of-file is encountered
- an exception is thrown

Returns an integer when used with no argument. When used with an argument, if a character is extracted, the get() function returns the this pointer. If no character is extracted setstate(failbit) is called. In any case a null char is appended to the array.

See Also

basic_istream::getline

Listing: Examples of basic_istream::get() usage:

// READ ONE CHARACTER:
// ewl-test file for input

// float 33.33 double 3.16e+10 Integer 789 character C
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
    using namespace std;
    char inFile[] = "ewl-test";
    ifstream in(inFile);
    if(!in.is_open())
    {cout << "Cannot open input file"; exit(1);} 
    char ch;
    while(in.get(ch)) cout << ch;
    return 0;
}

//float 33.33 double 3.16e+10 Integer 789 character C
// READ ONE LINE:

#include <iostream>
const int size = 100;
char buf[size];
int main()
{
    using namespace std;
    cout << " Enter your name: ";
    cin.get(buf, size);
    cout << buf;
    return 0;
}

Result:

Enter your name: Johnny Socksorter<enter>
Johnny Socksorter

18.2.4.3 basic_istream::getline

To obtain a delimiter terminated character sequence from an input stream.

    basic_istream<charT, traits>& getline(char_type* s,
    streamsize n, char_type delim = traits::newline());

Remarks
The unformatted `getline()` function retrieves character input, and stores it in a character array buffer if possible until the following conditions evaluated in this order occur. If no characters are extracted `setstate(failbit)` is called.

end-of-file occurs in which case `setstate(eofbit)` is called.

A delimiter (default value is the newline character) is encountered. In which case the delimiter is read and extracted but not stored.

A limit (the second argument minus one) is read.

If n-1 chars are read, that failbit gets set.

In any case it stores a null char into the next successive location of the array.

The this pointer is returned.

See Also

`basic_ostream::flush`

Listing: Example of `basic_istream::getline()` usage:

```cpp
#include <iostream>
const int size = 120;

int main()
{
    using namespace std;
    char compiler[size];
    cout << "Enter your compiler: ";
    cin.getline(compiler, size);
    cout << "You use " << compiler;
    return 0;
}
```

Result:

Enter your compiler:CodeWarrior <enter>
You use CodeWarrior

```cpp
#include <iostream>
const int size = 120;
#define TAB '	'

int main()
{
    using namespace std;
    cout << "What kind of Compiler do you use: ";
    char compiler[size];
    cin.getline(compiler, size,TAB);
    cout << compiler;
    cout << "second input not needed\n";
    cin >> compiler;
    cout << compiler;
    return 0;
}
```
Result:

What kind of Compiler do you use:

CodeWarrior	Why?

CodeWarrior

second input not needed

Why?

18.2.4.4 basic_istream::ignore

To extract and discard a number of characters.

```cpp
basic_istream<charT, traits>& ignore
  (streamsize n = 1, int_type delim = traits::eof());
```

Remarks

The function `ignore()` will extract and discard characters until:

- a limit is met (the first argument)
- end-of-file is encountered (in which case setstate(eofbit) is called)

The next character `c` is equal to the delimiter `delim`, in which case it is extracted except when `c` is equal to `traits::eof()`;

The this pointer is returned.

Listing: Example of basic_istream::ignore() usage:

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "ewl-test";
char bslash = '/';

int main()
```
```cpp
using namespace std;

ifstream in(inFile);
if(!in.is_open())
    {cout << "file not opened"; exit(1);}
char ch;
while((ch = in.get()) != EOF)
{
    if(ch == bslash && in.peek() == bslash)
    {
        in.ignore(100, '\n');
        cout << '\n';
    }
    else cout << ch;
}
return 0;
```

Result:

```cpp
char ch;
    /*This C comment will remain */
while((ch = in.get())!= EOF) cout.put(ch);
/* the C++ comments won't */
```

**18.2.4.5 basic_istream::peek**

To view the next character to be extracted.

```cpp
int_type peek();
```

**Remarks**

The function `peek()` allows you to look ahead at the next character in a stream to be extracted without extracting it.

If `good()` is false returns `traits::eof()` else returns the value of the next character in the stream.

**See Also**
Example of basic_istream::peek() usage see basic_istream::ignore

18.2.4.6  basic_istream::read

To obtain a block of binary data from an input stream.

    basic_istream<CharT, traits>& read
    (char_type* s, streamsize n);

Remarks

The function read() will attempt to extract a block of binary data until the following conditions are met.

A limit of n number of characters are stored.

end-of-file is encountered on the input (in which case setstate(failbit) is called.

Return

The this pointer is returned.

SeeAlso

basic_ostream::write

Listing: Example of basic_istream::read() usage:

```cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

struct stock {
    char name[80];
    double price;
    long trades;
};

char *Exchange = "BBSE";
char *Company = "Big Bucks Inc.";

int main()
{
```
using namespace std;

stock Opening, Closing;
strcpy(Opening.name, Company);
Opening.price = 180.25;
Opening.trades = 581300;

// open file for output
ofstream Market(Exchange, ios::out | ios::trunc | ios::binary);
if(!Market.is_open())
{cout << "can't open file for output"; exit(1);}
Market.write((char*) &Opening, sizeof(stock));
Market.close();

// open file for input
ifstream Market2(Exchange, ios::in | ios::binary);
if(!Market2.is_open())
{cout << "can't open file for input"; exit(2);}
Market2.read((char*) &Closing, sizeof(stock));

cout << Closing.name << "\n"
<< "The number of trades was: " << Closing.trades << '\n';
cout << fixed << setprecision(2)
<< "The closing price is: $" << Closing.price << endl;
Market2.close();
return 0;
}

Result:

Big Bucks Inc.
The number of trades was: 581300
The closing price is: $180.25

18.2.4.7 basic_istream::readsome

Extracts characters and stores them in an array.
streamsize readsome
  (charT_type* s, streamsize n);

Remarks

The function readsome extracts and stores characters in the buffer pointed to by s until the following conditions are met.

- end-of-file is encountered (in which case setstate(eofbit) is called.)
- no characters are extracted
- a limit of characters is extracted; either n or the size of the buffer.

Return

The number of characters extracted.

Listing: Example of basic_istream::readsome() usage.

The file ewl-test contains:
CodeWarrior
Software at Work
Registered Trademark

```c++
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>
const short SIZE = 81;
int main()
{
  using namespace std;
  ifstream in("ewl-test");
  if(!in.is_open())
    {cout << "can't open file for input"; exit(1);}
  char Buffer[SIZE] = \0;  
  ostringstream Paragraph;
  while(in.good() && (in.peek() != EOF))
    {
    in.readsome(Buffer, 5);
    Paragraph << Buffer;
    }
  cout << Paragraph.str();
  in.close();
  return 0;
}
```

Result:

CodeWarrior
Software at Work
Registered Trademark
18.2.4.8 basic_istream::putback

To replace a previously extracted character.

    basic_istream<charT, traits>& putback
           (char_type c);

Remarks

The function putback() allows you to replace the last character extracted by calling
rdbuf()->sungetc(). If the buffer is empty, or if sungetc() returns eof, setstate(failbit) may
be called.

Return

The this pointer is returned.

See Also

basic_istream::unget

Listing: Example of basic_istream::putback usage:

    // The file ewl-test contains.
    char ch; // to save char
    /* comment will remain */
    while((ch = in.get())!= EOF) cout.put(ch);
    // read until failure

    #include <iostream>
    #include <fstream>
    #include <stdlib.h>
    char inFile[] = "ewl-test";
    char bslash = '/';
    int main()
    {
        using namespace std;

        ifstream in(inFile);
        if(!in.is_open())
            {cout << "file not opened"; exit(1);}

        char ch, tmp;
        while((ch = in.get()) != EOF)
        {
            if(ch == bslash)
                { in.get(tmp);
                  if(tmp != bslash)
                      in.putback(tmp);
                  else continue;
                }
            cout << ch;
        return 0;
    }
Result:

```cpp
char ch;  // to save char
/* comment will remain */
while((ch = in.get()) != EOF) cout.put(ch);
read until failure
```

18.2.4.9 basic_istream::unget

To replace a previously extracted character.

```cpp
basic_istream<charT, traits>& unget();
```

Remarks

Use the function unget() to return the previously extracted character. If rdbuf() is null or if end-of-file is encountered setstate(badbit) is called.

The this pointer is returned.

See Also

basic_istream::putback , basic_istream::ignore

Listing: Example of basic_istream::unget() usage:

```cpp
// The file ewl-test contains:
// char ch;   // to save char
//              /* comment will remain */
//              // read until failure
// while((ch = in.get()) != EOF) cout.put(ch);

#include <iostream>
#include <fstream>
#include <cstdlib>
char inFile[] = "ewl-test";
char bslash = '/';

int main()
{
    using namespace std;
    ifstream in(inFile);
    if(!in.is_open())
    {
        cout << "file not opened"; exit(1);
    }
    char ch, tmp;
    while((ch = in.get()) != EOF)
    {
        if(ch == bslash)
        {
            in.get(tmp);
            if(tmp != bslash)
                in.unget();
            else continue;
        }
        cout << ch;
    }
    return 0;
}
```
return 0;
}

Result:

char ch;   // to save char
    /* comment will remain */
    // read until failure
while((ch = in.get()) != EOF) cout.put(ch);

18.2.4.10  basic_istream::sync

Synchronizes input and output

    int sync();

Remarks

This function attempts to make the input source consistent with the stream being extracted.

If rdbuf() -> pubsync() returns -1 setstate(badbit) is called and traits::eof is returned.

Return

If rdbuf() is Null returns -1 otherwise returns zero.

Listing: Example of basic_istream::sync() usage:

    // The file ewl-test contains:
    // This functions attempts to make the input source
    // consistent with the stream being extracted.
    // --
    // CodeWarrior "Software at Work"
#include <iostream>
#include <fstream>
#include <cstdlib>
char inFile[] = "ewl-test";
int main()
{
    using namespace std;
    ifstream in(inFile);
    if(!in.is_open())
    {
        cout << "could not open file"; exit(1);
    }
char str[10];
if(in.sync()) // return 0 if successful
  { cout << "cannot sync"; exit(1); }
while (in.good())
{
  in.get(str, 10, EOF);
  cout <<str;
}
return 0;

Result:
This functions attempts to make the input source
consistent with the stream being extracted.
--
CodeWarrior "Software at Work"

18.2.4.11 basic_istream::tellg

Determines the offset of the get pointer in a stream

pos_type tellg();

Remarks
The function tellg calls rdbuf()->pubseekoff(0, cur, in).
The current offset is a pos_type if successful, else returns -1.

See Also
basic_streambuf::pubseekoff()
Example of basic_istream::tellg() usage see basic_istream::seekg

18.2.4.12 basic_istream::seekg

Moves to a variable position in a stream.
basic_istream<charT, traits>& seekg(pos_type);

basic_istream<charT, traits>& seekg
(off_type, ios_base::seekdir dir);

Remarks

The function seekg is overloaded to take a pos_type object, or an off_type object (defined in basic_ios class.) The function is used to set the position of the get pointer of a stream to a random location for character extraction.

The this pointer is returned.

See Also

basic_streambuf::pubseekoff() and pubseekpos().

Listing: Example of basic_istream::seekg() usage:

```cpp
// The file ewl-test contains:
// ABCDEFGHIJKLMNOPQRSTUVWXYZ
#include <iostream>
#include <fstream>
#include <cstdlib>
int main()
{
    using namespace std;
    ifstream in("ewl-test");
    if(!in.is_open())
    {cout << "could not open file"; exit(1);}

    streampos spEnd(5), spStart(5);
    in.seekg(spStart);
    streampos aCheck = in.tellg();
    cout << "The offset at the start of the reading in bytes is "
        << aCheck << endl;
    char ch;
    while(spEnd != spStart+10)
    {
        in.get(ch);
        cout << ch;
        spEnd = in.tellg();
    }

    aCheck = in.tellg();
    cout << "\nThe current position's offset in bytes now is "
        << aCheck << endl;

    streamoff gSet = 0;
    in.seekg(gSet, ios::beg);
    aCheck = in.tellg();
    cout << "The final position's offset in bytes now is "
        << aCheck << endl;

    in.close();
    return 0;
}
```

Result:
18.2.5 Standard basic_istream manipulators

The istream class provides several manipulators for input streams.

18.2.5.1 basic_ifstream::ws

Provides inline style formatting.

```cpp
template<class charT, class traits>
basic_istream<charT, traits> &ws
(basic_istream<charT,traits>& is);
```

Remarks

The ws manipulator skips whitespace characters in input.

The this pointer is returned.

Listing: Example of basic_istream:: manipulator ws usage:

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
    char * inFileName = "ewl-test";
    ifstream in(inFileName);
    if (!in.is_open())
    {cout << "Couldn't open for input\n"; exit(1);}
    char ch;
    in.unsetf(ios::skipws);
```
cout << "Does not skip whitespace\n|";
while (1)
{
    in >> ch; // does not skip white spaces
    if (in.good())
        cout << ch;
    else break;
}
cout << "|\n\n";

//reset file position
in.clear();
in.seekg(0, ios::beg);

cout << "Does skip whitespace\n|";
while (1)
{
    in >> ws >> ch; // ignore white spaces
    if (in.good())
        cout << ch;
    else break;
}
cout << "|" << endl;

in.close();
return(0);

Result:

Does not skip whitespace
| a   b   c |
Does skip whitespace
| abc |
18.2.5.2 basic_iostream Constructor

Constructs and destroys an object of the class basic_iostream.

```cpp
explicit basic_iostream(basic_streambuf<charT, traits>* sb);
```

Remarks

Calls `basic_istream<charT, traits>(sb)` and `basic_ostream<charT, traits>(sb)`. After it is constructed `rdbuf()` equals `sb` and `gcount()` equals zero.

18.2.5.2.1 Destructor

```cpp
virtual ~basic_iostream();
```

Remarks

Destroys an object of type `basic_iostream`.

18.3 Output streams

The include file `<ostream>` includes classes and types that provide output stream mechanisms.

The topics in this section are:

- Template class basic_ostream
- Class basic_ostream::sentry
- Formatted output functions
- Unformatted output functions
- Standard basic_ostream manipulators

18.3.1 Template class basic_ostream

A class for stream output mechanisms.
The basic_ostream class provides for output stream mechanisms for output stream classes. The basic_ostream class may be used as an independent class, as a base class for the basic_ofstream class or a user derived class.

18.3.1.1 basic_ostream Constructor

Creates basic_ostream object for stream output.

```cpp
explicit basic_ostream(basic_streambuf<char_type, traits>*sb);
```

Remarks

The basic_ostream constructor constructs and initializes the base class object.

18.3.1.2 Destructor

Destroys an object of class basic_ostream.

```cpp
virtual ~basic_ostream();
```

Remarks

Removes a basic_ostream object from memory.

Listing: Example of basic_ostream() usage:

```cpp
// The ewl-test file contains originally
// CodeWarrior "Software at Work"
#include <iostream>
#include <fstream>
#include <cstdlib>
char inFile[] = "ewl-test";
int main()
{
using namespace std;

    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
    {
        cout << "Could not open file"; exit(1);
    }
```
ostream Out(inOut.rdbuf());
char str[] = "\nRegistered Trademark";
inOut.rdbuf()->pubseekoff(0, ios::end);
Out << str;
inOut.close();

return 0;
}

Result:

The File now reads:
CodeWarrior "Software at Work"
Registered Trademark

18.3.2 Class basic_ostream::sentry

A class for exception safe prefix and suffix operations.

18.3.2.1 Class basic_ostream::sentry Constructor

Prepare for formatted or unformatted output.

   explicit sentry(basic_ostream<charT, traits>& os);

Remarks

If after the operation os.good() is true ok_ equals true otherwise ok_ equals false. The constructor may call setstate(failbit) which may throw an exception.

18.3.2.2 Destructor

   ~sentry();

Remarks

The destructor under normal circumstances will call os.flush().
18.3.2.3 sentry::Operator bool

Returns the value of the data member ok_.

operator bool();

Remarks
Operator bool returns the value of ok_.

18.3.3 Formatted output functions

Formatted output functions provide a manner of inserting for output specific data types.

18.3.3.1 Common requirements

The operations begin by calling opfx() and end by calling osfx() then returning the value specified for the formatted output.

Some output maybe generated by converting the scalar data type to a NTBS (null terminated byte string) text.

If the function fails for any reason the function calls set state(failbit).

18.3.3.2 Arithmetic Inserter Operator <<

Provides formatted insertion of types into a stream.

basic_ostream<charT, traits>& operator<<(short n)
basic_ostream<charT, traits>& operator<<(unsigned short n)
basic_ostream<charT, traits>& operator<<(int n)
basic_ostream<charT, traits>& operator<<(unsigned int n)
basic_ostream<charT, traits>& operator<<(long n)
basic_ostream<charT, traits>& operator<<(unsigned long n)
basic_ostream<charT, traits>& operator<<(float f)
basic_ostream<charT, traits>& operator<<(double f)
basic_ostream<charT, traits>& operator<<(long double f)

Remarks
Converts an arithmetic value. The formatted values are converted as if they had the same behavior of the `fprintf()` function.

In most cases this is returned unless failure, in which case set `state(failbit)` is called.

### Table 18-2. Output states and stdio equivalents.

<table>
<thead>
<tr>
<th>Output State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td></td>
</tr>
<tr>
<td>(flags() &amp; basefield) == oct</td>
<td>%o</td>
</tr>
<tr>
<td>(flags() &amp; basefield) == hex</td>
<td>%x</td>
</tr>
<tr>
<td>(flags() &amp; basefield) != 0</td>
<td>%x</td>
</tr>
<tr>
<td>Otherwise</td>
<td>signed integral type</td>
</tr>
<tr>
<td></td>
<td>unsigned integral type</td>
</tr>
<tr>
<td>Floating Point Numbers</td>
<td></td>
</tr>
<tr>
<td>(flags() &amp; floatfield) == fixed</td>
<td>%f</td>
</tr>
<tr>
<td>(flags() &amp; floatfield) == scientific (flags() &amp; uppercase) != 0</td>
<td>%e %E</td>
</tr>
<tr>
<td>Otherwise</td>
<td>(flags() &amp; uppercase) != 0</td>
</tr>
<tr>
<td></td>
<td>An integral type other than a char type</td>
</tr>
<tr>
<td></td>
<td>(flags() &amp; showpos) != 0 (flags() &amp; showbase) != 0</td>
</tr>
<tr>
<td></td>
<td>A floating point type</td>
</tr>
<tr>
<td></td>
<td>(flags() &amp; showpos) != 0 (flags() &amp; showpoint) != 0</td>
</tr>
</tbody>
</table>

For any conversion, if `width()` is non-zero then a field with a conversion specification has the value of `width()`.

For any conversion, if (flags() and fixed) !=0 or if precision() >0 the conversion specification is the value of precision().

For any conversion, padding behaves in the following manner.

### Table 18-3. Conversion state and stdio equivalents

<table>
<thead>
<tr>
<th>State</th>
<th>Justification</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(flags()&amp; adjustfield) == left</td>
<td>left</td>
<td>space padding</td>
</tr>
<tr>
<td>(flags() &amp; adjustfield) == internal</td>
<td>Internal</td>
<td>zero padding</td>
</tr>
<tr>
<td>Otherwise</td>
<td>right</td>
<td>space padding</td>
</tr>
</tbody>
</table>

The `ostream` insertion operators are overloaded to provide for insertion of most predefined types into an output stream. They return a reference to the `basic stream` object so they may be used in a chain of statements to input various types to the same stream.
18.3.3.3 basic_ostream::operator<<

basic_ostream<charT, traits>& operator<<
(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&));

Returns pf(*this).

basic_ostream<charT, traits>& operator<<
(basic_ostream<charT, traits>& (*pf)(basic_ios<charT, traits>&));

Calls pf(*this) return *this.

basic_ostream<charT, traits>& operator<<(const char_type *s)
basic_ostream<charT, traits>& operator<<(char_type c)
basic_ostream<charT, traits>& operator<<(bool n)

Behaves depending on how the boolalpha flag is set.

basic_ostream<charT, traits>& operator<<(void p)

Converts the pointer to void p as if the specifier was %p and returns *this.

basic_ostream<charT, traits>& operator<<(basic_streambuf<char_type, traits>* sb);

If sb is null calls setstate(failbit) otherwise gets characters from sb and inserts them into *this until:
   • end-of-file occurs
   • inserting into the stream fails
   • an exception is thrown.

If the operation fails, it calls setstate(failbit) or re-throws the exception, otherwise returns *this.

Remarks

The formatted output functions insert the values into the appropriate argument type. Most inserters (unless noted otherwise) return the this pointer.

Listing: Example of basic_ostream inserter usage:
```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

char oFile[81] = "ewl-test";

int main()
{
    using namespace std;

    ofstream out(oFile);
    out << "float " << 33.33;
    out << " double " << 3.16e+10;
    out << " Integer " << 789;
    out << " character " << 'C' << endl;
    out.close();
    cout << "float " << 33.33;
    cout << " double " << 3.16e+10;
    cout << " Integer " << 789;
    cout << " character " << 'C' << endl;
    return 0;
}

Result:

Output: to ewl-test
float 33.33 double 3.16e+10 Integer 789 character C
Output to console
float 33.33
double 3.16e+10
Integer 789
character C

18.3.3.4 Overloading Inserters

Provides specialized output mechanisms for an object.

Overloading
inserter prototype
basic_ostream &operator<<(basic_ostream &stream, const omanip<T>&)

Remarks
You may overload the inserter operator to tailor it to the specific needs of a particular class.

The this pointer is returned.

Listing: Example of overloaded inserter usage:

```cpp
#include <iostream>
#include <string.h>
#include <iomanip>

class phonebook {
  friend ostream &operator<<(ostream &stream, phonebook o);

protected:
  char *name;
  int areacode;
  int exchange;
  int num;

public:
  phonebook(char *n, int a, int p, int nm) :
    areacode(a),
    exchange(p),
    num(nm),
    name(n) {}
};

int main()
{
  using namespace std;
  phonebook a("Sales", 800, 377, 5416);
  phonebook b("Voice", 512, 873, 4700);
  phonebook c("Fax", 512, 873, 4900);
  cout << a << b << c;
  return 0;
}

std::ostream &operator<<(std::ostream &stream, phonebook o)
{
  stream << o.name << " ";
```
stream << "(" << o.areacode << ") ";
stream << o.exchange << "-";
stream << setfill('0') << setw(4)
    << o.num << "\n";
return stream;

Result:

Sales (800) 377-5416
Voice (512) 873-4700
Fax (512) 873-4900

18.3.4 Unformatted output functions

Each unformatted output function begins by creating an object of the class sentry. The unformatted output functions are ended by destroying the sentry object and may return a value specified.

18.3.4.1 basic_ostream::tellp

Returns the offset of the put pointer in an output stream.

pos_type tellp();

Return

If fail() returns -1 else returns rdbuf()->pubseekoff(0, cur, out).

See Also

basic_istream::tellg, basic_ostream::seekp

Example of basic_ostream::tellp() usage see basic_ostream::seekp

18.3.4.2 basic_ostream::seekp

Randomly move to a position in an output stream.
basic_ostream<charT, traits>& seekp(pos_type);

basic_ostream<charT, traits>& seekp
    (off_type, iosbase::seekdir);

Remarks

The function seekp is overloaded to take a single argument of a pos_type pos that calls rdbuf()->pubseekpos(pos). It is also overloaded to take two arguments: an off_type off and i os_base::seekdir type dir that calls rdbuf()->pubseekoff(off, dir).

Returns the this pointer.

See Also

basic_istream::seekg , basic_ostream::tellp

Listing: Example of basic_ostream::seekp() usage.

#include <iostream>
#include <sstream>
#include <string>

std::string motto = "CodeWarrior - Software at Work";

int main()
{
    using namespace std;
    ostringstream ostr(motto);
    streampos cur_pos, start_pos;
    cout << "The original array was :
" << motto << "\n"
    << motto << "\n"
    // associate buffer
    stringbuf *strbuf(ostr.rdbuf());
    streamoff str_off = 10;
    cur_pos = ostr.tellp();
    cout << "The current position is "
    << cur_pos.offset()
    << " from the beginning\n";
    ostr.seekp(str_off);
    cur_pos = ostr.tellp();
    cout << "The current position is "
    << cur_pos.offset()
    << " from the beginning\n";
    strbuf->sputc('0');
    cout << "The stringbuf array is\n" << strbuf->str() << "\n"
    cout << "The ostringstream array is still\n" << motto;
    return 0;
}

Results:

The original array was :

CodeWarrior - Software at Work

The current position is 0 from the beginning
The current position is 10 from the beginning

The stringbuf array is
CodeWarrior

The ostringstream array is still
CodeWarrior - Software at Work

18.3.4.3 basic_ostream::put

Places a single character in the output stream.

    basic_ostream<charT, traits>& put(char_type c);

Remarks

The unformatted function put() inserts one character in the output stream. If the operation fails, it calls setstate(badbit).

The this pointer is returned.

Listing: Example of basic_ostream::put() usage:

    #include <iostream>
    int main()
    {
        using namespace std;

        char *str = "CodeWarrior \"Software at Work\"";
        while(*str)
        {
            cout.put(*str++);
        }
        return 0;
    }

Result:

    CodeWarrior "Software at Work"
18.3.4.4 basic_ostream::write

Inserts a block of binary data into an output stream.

basic_ostream<CharT, Traits>& write
(const char_type* s, streamsize n);

Remarks

The overloaded function write() is used to insert a block of binary data into a stream. This function can be used to write an object by casting that object as a unsigned char pointer. If the operation fails, setstate(badbit) is called.

A reference to ostream. The this pointer is returned.

SeeAlso

basic_istream::read

Listing: Example of basic_ostream::write() usage:

```cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

struct stock {
    char name[80];
    double price;
    long trades;
};

char *Exchange = "BBSE";
char *Company = "Big Bucks Inc.";

int main() {
    using namespace std;
    stock Opening, Closing;
    strcpy(Opening.name, Company);
    Opening.price = 180.25;
    Opening.trades = 581300;
    // open file for output
    ofstream Market(Exchange, ios::out | ios::trunc | ios::binary);
    if(!Market.is_open()) {
        cout << "can't open file for output"; exit(1);
    }
    Market.write((char*) &Opening, sizeof(stock));
    Market.close();
    // open file for input
    ifstream Market2(Exchange, ios::in | ios::binary);
    if(!Market2.is_open()) {
        cout << "can't open file for input"; exit(2);
    }
```
Market2.read((char*) &Closing, sizeof(stock));
cout << Closing.name << "\n"
   << "The number of trades was: "
   << Closing.trades << '\n';

cout << fixed << setprecision(2)
   << "The closing price is: $" 
   << Closing.price << endl;

Market2.close();
return 0;
}

Result:

Big Bucks Inc.
The number of trades was: 581300
The closing price is: $180.25

18.3.4.5 basic_ostream::flush

Forces the output buffer to release its contents.

    basic_ostream<CharT, Traits>& flush();

Remarks

The function flush() is an output only function in C++. You may use it for an immediate expulsion of the output buffer. This is useful when you have critical data or you need to ensure that a sequence of events occurs in a particular order. If the operation fails, it calls setstate(badbit).

The this pointer is returned.

Note that in the Example of basic_ostream::flush() usage: if you comment out the flush both lines will display simultaneously at the end of the program.

Listing: Example of basic_ostream::flush() usage:

    #include <iostream>
    #include <iomanip>
    #include <ctime>
    class stopwatch {
    private:
        double begin, set, end;
    public:
        stopwatch();
```cpp
-stopwatch();
void start();
void stop();
};
stopwatch::stopwatch()
{
  using namespace std;

  begin = (double) clock() / CLOCKS_PER_SEC;
  end = 0.0;
  start();
  cout << "begin the timer: ";
}
stopwatch::~stopwatch()
{
  using namespace std;
  stop(); // set end
  cout << "\nThe Object lasted: ";
  cout << fixed << setprecision(2)
       << end - begin << " seconds \n";
}
// clock ticks divided by ticks per second
void stopwatch::start()
{
  using namespace std;
  set = double(clock()/CLOCKS_PER_SEC);
}
void stopwatch::stop()
{
  using namespace std;
  end = double(clock()/CLOCKS_PER_SEC);
}
void time_delay(unsigned short t);
int main()
{

```
```
using namespace std;

    stopwatch watch; // create object and initialize
    cout.flush(); // this flushes the buffer
    time_delay(5);
    return 0; // destructor called at return

} //time delay function

void time_delay(unsigned short t)
{
    using namespace std;
    time_t tStart, tEnd;
    time(&tStart);
    time(&tStart);
    while(tStart + t > time(&tEnd)){};

} Result:

    begin the timer: < immediate display then pause >
    begin the timer:
    The Object lasted: 3.83 seconds

18.3.5 Standard basic_ostream manipulators

The ostream class provides an inline formatting mechanism.

18.3.5.1 basic_ostream:: endl

To insert a newline and flush the output stream.

    template < class charT, class traits >
    basic_ostream<charT, traits> & endl
    (basic_ostream<charT,traits>& os);

Remarks

The manipulator endl takes no external arguments, but is placed in the stream. It inserts a newline character into the stream and flushes the output.
A reference to basic_ostream. The this pointer is returned.

See Also

basic_ostream::operator

18.3.5.2 basic_ostream::ends

To insert a NULL character.

```
template< class charT, class traits >
basic_ostream<charT, traits> &ends
(basic_ostream<charT,traits>& os);
```

Remarks

The manipulator ends, takes no external arguments, but is placed in the stream. It inserts a NULL character into the stream, usually to terminate a string.

A reference to ostream. The this pointer is returned.

The ostringstream provides in-core character streams but must be null terminated by the user. The manipulator ends provides a null terminator.

Listing: Example of basic_ostream::ends usage:

```
#include <iostream>
#include <sstream>

int main()
{
    using namespace std;

    ostringstream out;  // see note above
    out << "Ask the teacher anything\n";
    out << "OK, what is 2 + 2?\n";
    out << 2 << " plus " << 2 << " equals 
               << 4 << ends;
    cout << out.str();
    return 0;
}
```

Result:
Ask the teacher anything
OK, what is 2 + 2?
2 plus 2 equals 4?

18.3.5.3 basic_ostream::flush

To flush the stream for output.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits> &
flush(basic_ostream<charT, traits> &os);
```

Remarks

The manipulator `flush`, takes no external arguments, but is placed in the stream. The manipulator `flush` will attempt to release an output buffer for immediate use without waiting for an external input.

A reference to `ostream`. The `this` pointer is returned.

Note in the Example of `basic_ostream::flush` usage: comment out the flush and both lines will display simultaneously at the end of the program.

See Also

`basic_ostream::flush`

Listing: Example of `basic_ostream::flush` usage:

```cpp
#include <iostream>
#include <iomanip>
#include <ctime>

class stopwatch {
private:
  double begin, set, end;
public:
  stopwatch();
  ~stopwatch();
  void start();
  void stop();
};

stopwatch::stopwatch()
{
  using namespace std;
  begin = (double) clock() / CLOCKS_PER_SEC;
  end   = 0.0;

  start();
  {
    begin = (double) clock() / CLOCKS_PER_SEC;
    end   = 0.0;
    start();
    cout << "begin time the timer: " << flush;
  }
```
stopwatch::~stopwatch()
{
    using namespace std;
    stop();    // set end
    cout << "The Object lasted: ";
    cout << fixed << setprecision(2)
    << end - begin << " seconds \n";
}

// clock ticks divided by ticks per second
void stopwatch::start()
{
    using namespace std;
    set = double(clock()/CLOCKS_PER_SEC);
}

void stopwatch::stop()
{
    using namespace std;
    end = double(clock()/CLOCKS_PER_SEC);
}

void time_delay(unsigned short t);

int main()
{
    using namespace std;
    stopwatch watch; // create object and initialize
time_delay(5);
    return 0; // destructor called at return
}

// time delay function
void time_delay(unsigned short t)
{
    using namespace std;
    time_t tStart, tEnd;
time(&tStart);
time(&tStart);
    while(tStart + t > time(&tEnd)){}
}

Results:

    begin time the timer:
    < short pause >
    The Object lasted: 3.78 seconds

18.4 Standard manipulators

The include file iomanip defines a template class and related functions for input and output manipulation.

18.4.1 Standard Manipulator Instantiations

Creates a specific use instance of a template by replacing the parameterized elements with pre-defined types.
18.4.2 resetiosflags

To unset previously set formatting flags.

Prototypes

smanip resetiosflags(ios_base::fmtflags mask)

Remarks

Use the manipulator resetiosflags directly in a stream to reset any format flags to a previous condition. You would normally store the return value of setf() in order to achieve this task.

A smanip type is returned, which is an implementation defined type.

See Also

ios_base::setf(), ios_base::unsetf()

Listing: Example of resetiosflags() usage:

```cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    double d = 2933.51;
    long flags;
    flags = ios::scientific | ios::showpos | ios::showpoint;

    cout << "Original: " << d << endl;
    cout << "Flags set: " << setiosflags(flags) << d << endl;
    cout << "Flags reset to original: "
         << resetiosflags(flags) << d << endl;
    return 0;
}
```

Result:

Original:   2933.51
Flags set:   +2.933510e+03
Flags reset to original:   2933.51

18.4.3 setiosflags

...
To set the stream format flags.

**Prototypes**

```cpp
smanip setiosflags(ios_base::fmtflags mask)
```

**Remarks**

Use the manipulator `setiosflags()` to set the input and output formatting flags directly in the stream.

A `smanip` type is returned, which is an implementation defined type.

**See Also**

`ios_base::setf()`, `ios_base::unsetf()`

For example of `setiosflags()` usage see `resetiosflags`

### 18.4.4 setbase

To set the numeric base of an output.

```cpp
smanip setbase(int)
```

**Remarks**

The manipulator `setbase()` directly sets the numeric base of integral output to the stream. The arguments are in the form of 8, 10, 16, or 0, and 8 octal, 10 decimal and 16 hexadecimal. Zero represents `ios::basefield`; a combination of all three.

Returns a `smanip` type, which is an implementation defined type.

**See Also**

`ios_base::setf()`

**Listing: Example of setbase usage:**

```cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    cout << "Hexadecimal " << setbase(16) << 196 << 'n';
    cout << "Decimal " << setbase(10)      << 196 << 'n';
    cout << "Octal  " << setbase(8)       << 196 << 'n';
    cout.setf(ios::hex, ios::oct | ios::hex);
}
```
cout << "Reset to Hex " << 196 << '\n';
cout << "Reset basefield setting "
    << setbase(0) << 196 << endl;
return 0;
}

Result:

Hexadecimal c4
Decimal 196
Octal 304
Reset to Hex c4
Reset basefield setting 196

18.4.5 setfill

To specify the characters to insert in unused spaces in the output.

smanip setfill(int c)

Remarks

Use the manipulator setfill() directly in the output to fill blank spaces with character c.

Returns a smanip type, which is an implementation defined type.

See Also

basic_ios::fill

Listing: Example of basic_ios::setfill() usage:

#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    cout.width(8);
    cout << setfill('*') << "Hi!" << '\n';
    char fill = cout.fill();
    cout << "The filler is a " << fill << endl;
    return 0;
}

Result:

Hi!*****
The filler is a *

18.4.6 setprecision
Set and return the current format precision.

`smanip<int> setprecision(int)`

**Remarks**

Use the manipulator `setprecision()` directly in the output stream with floating point numbers to limit the number of digits. You may use `setprecision()` with scientific or non-scientific floating point numbers.

With the flag `ios::floatfield` set, the number in `setprecision` refers to the total number of significant digits generated. If the settings are for either `ios::scientific` or `ios::fixed` then the `setprecision` refers to the number of digits after the decimal place.

This means that `ios::scientific` will have one more significant digit than `ios::floatfield`, and `ios::fixed` will have a varying number of digits.

Returns a `smanip` type, which is an implementation defined type.

**See Also**

`ios_base::setf()`, `ios_base::precision()`

**Listing: Example of setprecision() usage:**

```cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    cout << "Original: " << 321.123456 << endl;
    cout << "Precision set: " << setprecision(8) << 321.123456 << endl;
    return 0;
}
```

**Result:**

```
Original: 321.123
Precision set: 321.12346
```

**18.4.7 setw**

To set the width of the output field.

`smanip<int> setw(int)`

**Remarks**
Use the manipulator `setw()` directly in a stream to set the field size for output. A pointer to `ostream` is returned.

**See Also**

`ios_base::width()`

**Listing: Example of `setw()` usage:**

```cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    cout << setw(8)
         << setfill('*')
         << "Hi!" << endl;
    return 0;
}
```

**Result:**

```
Hi!*****
```

### 18.4.8 Overloaded Manipulator

To store a function pointer and object type for input.

Overloaded input manipulator for `int` type.

```cpp
istream &imanip_name(istream &stream, int param) {
    // body of code
    return stream;
}
```

Overloaded output manipulator for `int` type.

```cpp
ostream &omanip_name(ostream &stream, int param) {
    // body of code
    return stream;
}
```

For other input/output types

```cpp
smanip<type> mainip_name(type param) {
    return smanip<type> (manip_name, param);
}
```

**Remarks**

Use an overloaded manipulator to provide special and unique input handling characteristics for your class.

Returns a pointer to stream object.

**Listing: Example of overloaded manipulator usage:**
```cpp
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cctype>

char buffer[80];
char *Password = "CodeWarrior";

struct verify
{
    explicit verify(char* check) : check_(check) {}
    char* check_;
};

char *StrUpr(char * str);
std::istream& operator >> (std::istream& stream, const verify& v);

int main()
{
    using namespace std;
    cin >> verify(StrUpr(Password));
    cout << "Log in was Completed ! \n";
    return 0;
}

std::istream& operator >> (std::istream& stream, const verify& v)
{
    using namespace std;
    short attempts = 3;
    do {
        cout << "Enter password: ";
        stream >> buffer;
        StrUpr(buffer);

        if (! strcmp(v.check_, buffer)) return stream;
        cout << "\a\a";
        attempts--;
    } while(attempts > 0);
    cout << "All Tries failed \n";
    exit(1);
    return stream;
}

char *StrUpr(char * str)
{
    char *p = str; // dupe string
    while(*p) *p++ = static_cast<char>(std::toupper(*p));
    return str;
}

Result:

Enter password: <codewarrior>
Enter password: <mw>
Enter password: <CodeWarrior>
Log in was Completed !
```
Chapter 19
String Based Streams

This chapter discusses string-based streams in the standard C++ library.

There are four template classes and 6 various types defined in the header <sstream> that are used to associate stream buffers with objects of class basic_string.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Header <sstream>
- Template class basic_stringbuf
- Template class basic_istringstream
- Class basic_ostringstream
- Class basic_stringstream

19.1 Header <sstream>

The header <sstream> includes classes and types that associate stream buffers with string objects for input and output manipulations.

NOTE
The class basic_string is discussed in previous chapters.

19.2 Template class basic_stringbuf

The template class basic_stringbuf is derived from basic_streambuf and used to associate both input and output streams with an object of class basic_string.
The class `basic_stringbuf` is derived from `basic_streambuf` to associate a stream with a `basic_string` object for in-core memory character manipulations.

### 19.2.1 `basic_stringbuf` constructors

The `basic_stringbuf` has two constructors to create a string buffer for characters for input/output.

```cpp
explicit basic_stringbuf(ios_base::openmode which = ios_base::in | ios_base::out);
explicit basic_stringbuf(const basic_string<char_type> &str, ios_base::openmode which = ios_base::in | ios_base::out);
```

**Remarks**

The `basic_stringbuf` constructor is used to create an object usually as an intermediate storage object for input and output. The overloaded constructor is used to determine the input or output attributes of the `basic_string` object when it is created.

No array object is allocated for the first `basic_stringbuf` constructor.

**Listing: Example of `basic_stringbuf::basic_stringbuf()` usage:**

```cpp
#include <iostream>
#include <sstream>

const int size = 100;
int main()
{
    using namespace std;

    stringbuf strbuf;
    strbuf.pubsetbuf('\0', size);
    strbuf.sputn("ABCDE", 50);
    char ch;
    // look ahead at the next character
    ch = strbuf.snextc();
    cout << ch;
    // get pointer was not returned after peeking
    ch = strbuf.snextc();
    cout << ch;
```
19.2.2 Member functions

The class basic_stringbuf has one member function.

19.2.2.1 basic_stringbuf::str

To return or clear the basic_string object stored in the buffer.

```cpp
basic_string<char_type> str() const;
void str(const basic_string<char_type>& s);
```

Remarks

The function `str()` freezes the buffer then returns a basic_string object.

The function `str(const string s)` assigns the value of the string `s` to the stringbuf object.

The no argument version returns a basic_string if successful. The function with an argument has no return.

Listing: Example of basic_stringbuf::str() usage:

```cpp
#include <iostream>
#include <sstream>
#include <cstring>
char CW[] = "CodeWarrior";
char AW[] = " - " Software at Work""

int main()
{
    using namespace std;
    stringbuf strbuf(buf, ios::in | ios::out);
    int size;
    size = strlen(CW);
    strbuf.sputn(CW, size);
    size = strlen(AW);
    strbuf.sputn(AW, size);

    cout << strbuf.str() << endl;

    // Clear the buffer then fill it with
    // new information and then display it
    string clrBuf = "";
```
```cpp
string ANewLine = "We Listen we Act";
strbuf.str(cirBuf);
strbuf.sputn(ANewLine.c_str(), ANewLine.size());
cout << strbuf.str() << endl;
return 0;
}
```

Results

CodeWarrior - "Software at Work"

We Listen we Act

### 19.2.3 Overridden virtual functions

The base class `basic_streambuf` has several virtual functions that are to be overloaded by derived classes. They are:

- `underflow()``
- `pbackfail()``
- `overflow()``
- `seekoff()``
- `seekpos()``

#### 19.2.3.1 basic_stringbuf::underflow

To show an underflow condition and not increment the get pointer.

```cpp
virtual int_type underflow();
```

**Remarks**

The function `underflow` overrides the `basic_streambuf` virtual function.

Returns the first character of the pending sequence and does not increment the get pointer. If the position is `null` returns `traits::eof()` to indicate failure.

**See Also**

`basic_streambuf::underflow()`

#### 19.2.3.2 basic_stringbuf::pbackfail

Template class `basic_stringbuf`

EWL C++ Library Reference Manual, Rev. 10.x, 02/2014
To show a failure in a put back operation.

```cpp
virtual int_type pbackfail
    (int_type c = traits::eof());
```

**Remarks**

The function `pbackfail` overrides the basic_streambuf virtual function.

The function `pbackfail()` is only called when a put back operation really has failed and returns `traits::eof`. If success occurs the return is undefined.

**See Also**

`basic_streambuf::pbackfail()`

---

19.2.3.3 **basic_stringbuf::overflow**

Consumes the pending characters of an output sequence.

```cpp
virtual int_type overflow
    (int_type c = traits::eof());
```

**Remarks**

The function `overflow` overrides the basic_streambuf virtual function.

The function returns `traits::eof()` for failure or some unspecified result to indicate success.

**See Also**

`basic_streambuf::overflow()`

---

19.2.3.4 **basic_stringbuf::seekoff**

To return an offset of the current pointer in an input or output stream.

```cpp
virtual pos_type seekoff
    (off_type off,
     ios_base::seekdir way,
     ios_base::openmode which =
     ios_base::in | ios_base::out);
```

**Remarks**

The function `seekoff` overrides the basic_streambuf virtual function.
A `pos_type` value is returned, which is an invalid stream position.

**See Also**

`basic_streambuf::seekoff()`

### 19.2.3.5 `basic_stringbuf::seekpos`

To alter an input or output stream position.

```cpp
virtual pos_type seekpos
(pos_type sp,
 ios_base::openmode which =
 ios_base::in | ios_base::out);
```

**Remarks**

If the open mode is in or out, the function alters the stream position of both the input and output sequences. If the open mode is in, it alters the stream position of the input sequence. If the open mode is out, it alters the stream position of the output sequence. If `sp` is an invalid stream position, the operation fails and the return value is `pos_type(off_type(-1))`. Otherwise, the function returns the current new position.

If neither the in or out sequence is positioned, `pos_type(off_type(-1))` is returned.

**See Also**

`basic_streambuf::seekpos()`

### 19.3 Template class `basic_istringstream`

The template class `basic_istringstream` is derived from `basic_istream` and is used to associate input streams with an object of class `basic_string`.

The class `basic_istringstream` uses an object of type `basic_stringbuf` to control the associated storage.

### 19.3.1 `basic_istringstream` Constructor
The basic_istringstream constructors create a basic_stringstream object and initializes the basic_streambuf object.

```cpp
global implicit basic_istringstream (ios_base::openmode which = ios_base::in);

global implicit basic_istringstream (const basic_string<charT> &str, ios_base::openmode which =
ios_base::in);

**Remarks**

The basic_istringstream constructor is overloaded to accept an object of class basic_string for input.

**See Also**

basic_ostringstream, basic_stringstream

**Listing: Example of basic_istringstream::basic_istringstream() usage**

```cpp
#include <iostream>
#include <string>
#include <sstream>

int main()
{
    using namespace std;
    string sBuffer = "3 12.3 line";
    int num = 0;
    double flt = 0;
    char szArr[20] = "\0";
    "istringstream Paragraph(sBuffer, ios::in);
    Paragraph >> num;
    Paragraph >> flt;
    Paragraph >> szArr;

    cout << num << " " << flt << " "
    << szArr << endl;
    return 0;
}
```

**Result**

```
3 12.3 line
```

**19.3.2 Member functions**

The class basic_istringstream has two member functions.

**19.3.2.1 basic_istringstream::rdbuf**

To retrieve a pointer to the stream buffer.
template class basic_istringstream

basic_stringbuf<charT, traits>* rdbuf() const;

Remarks

To manipulate a stream for random access or synchronization it is necessary to retrieve a
pointer to the streams buffer. The function rdbuf() allows you to retrieve this pointer.

A pointer to an object of type basic_stringbuf sb is returned by the rdbuf function.

See Also

basic_ostringstream::rdbuf()

basic_ios::rdbuf()

basic_stringstream::rdbuf()

Listing: Example of basic_istringstream::rdbuf() usage.

#include <iostream>
#include <sstream>

std::string buf = "CodeWarrior - \"Software at work\"";
char words[50];

int main()
{
    using namespace std;
    istringstream ist(buf);
    istream in(ist.rdbuf());
    in.seekg(25);
    in.get(words,50);
    cout << words;
    return 0
}

Result

"Software at work"

19.3.2.2 basic_istringstream::str

To return or assign the basic_string object stored in the buffer.

basic_string<charT> str() const;
void str(const basic_string<charT> &s);

Remarks

The function str() freezes the buffer then returns a basic_string object.
The function str(const string s) assigns the value of the string `s` to the stringbuf object.
The no argument version returns a `basic_string` if successful. The function with an argument has no return.

**See Also**

- `basic_stringbuf::str()`
- `basic_ostringstream.str()`
- `basic_stringstream::str()`

**Listing: Example of basic_istringstream::str() usage.**

```cpp
#include <iostream>
#include <sstream>

std::string buf = "CodeWarrior - "Software at Work";"

int main()
{
    using namespace std;
    istringstream istr(buf);
    cout << istr.str();
    return 0;
}
```

**Result:**

```
CodeWarrior - "Software at Work"
```

### 19.4 Class basic_ostringstream

The template class `basic_ostringstream` is derived from `basic_ostream` and used to associate output streams with an object of class `basic_string`.

The class `basic_ostringstream` uses an object of type `basic_stringbuf` to control the associated storage.

#### 19.4.1 basic_ostringstream Constructor

The `basic_ostringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

```cpp
explicit basic_ostringstream
    (ios_base::openmode which = ios_base::out);

explicit basic_ostringstream
    (const basic_string<charT>& str,  
      ios_base::openmode which = ios_base::out);
```
Remarks

The basic_ostringstream constructor is overloaded to accept an object of class basic_string for output.

See Also

basic_istringstream, basic_stringstream

Listing: Example of basic_ostringstream::basic_ostringstream() usage

```cpp
// The file ewl-test contains
// CodeWarrior - "Software at Work"
// Registered Trademark
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>
int main()
{
    using namespace std;
    ifstream in("ewl-test");
    if(!in.is_open())
        {cout << "can't open file for input"; exit(1);}
    ostringstream Paragraph;
    char ch = '\0';
    while((ch = in.get()) != EOF)
    {
        Paragraph << ch;
    }
    cout << Paragraph.str();
    in.close();
    return 0;
}
```

Result:

```
CodeWarrior - "Software at Work"
Registered Trademark
```

19.4.2 Member functions

The class basic_ostringstream has two member functions.

19.4.2.1 basic_ostringstream::rdbuf

To retrieve a pointer to the stream buffer.

```
basic_stringbuf<charT, traits>* rdbuf() const;
```
Remarks

To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function \texttt{rdbuf()} allows you to retrieve this pointer.

A pointer to an object of type \texttt{basic_stringbuf sb} is returned by the \texttt{rdbuf} function.

See Also

\texttt{basic_ostringstream::rdbuf()}

\texttt{basic_ios::rdbuf()}

\texttt{basic_stringstream::rdbuf()}

Listing: example of basic_ostringstream::rdbuf() usage

```cpp
#include <iostream>
#include <sstream>
#include <string>

std::string motto = "CodeWarrior - "Software at Work"";

int main()
{
    using namespace std;
    ostringstream ostr(motto);
    streampos cur_pos(0), start_pos(0);
    cout << "The original array was:\n"
         << motto << "\n"
         << static_cast<streampos>(cur_pos);
    // associate buffer

    stringbuf *strbuf(ostr.rdbuf());
    streamoff str_off = 10;
    cur_pos = ostr.tellp();
    cout << "The current position is "
         << static_cast<streamoff>(cur_pos);
    cout << " from the beginning\n"

    ostr.seekp(str_off);
    cur_pos = ostr.tellp();
    cout << "The current position is "
         << static_cast<streamoff>(cur_pos);
    cout << " from the beginning\n"

    strbuf->sputc('0');
    cout << "The stringbuf array is\n"
         << strbuf->str() << "\n"

    cout << "The ostringstream array is still\n"
         << motto;
    return 0;
}
```

Results:

The original array was:
CodeWarrior - "Software at Work"
The current position is 0 from the beginning
The current position is 10 from the beginning

The stringbuf array is
CodeWarrior
CodeWarrior - "Software at Work"
19.4.2.2 basic_ostringstream::str

To return or assign the `basic_string` object stored in the buffer.

```cpp
basic_string<CharT> str() const;
void str(const basic_string<CharT> &s);
```

Remarks

The function `str()` freezes the buffer then returns a `basic_string` object.

The function `str(const string s)` assigns the value of the string `s` to the `stringbuf` object.

The no argument version returns a `basic_string` if successful. The function with an argument has no return.

See Also

basic_stringbuf::str(), basic_istringstream.str()

basic_stringstream::str()

Listing: Example of basic_ostringstream::str() usage.

```cpp
#include <iostream>
#include <sstream>

int main()
{
    using namespace std;
    ostringstream out;
    out << "Ask the teacher anything\n";
    out << "OK, what is 2 + 2?\n";
    out << 2 << " plus " << 2 << " equals "
        << 4 << ends;
    cout << out.str();
    return 0;
}
```

Result:

```
Ask the teacher anything
OK, what is 2 + 2?
2 plus 2 equals 4?
```

19.5 Class basic_stringstream
The template class `basic_stringstream` is derived from `basic_iostream` and used to associate input and output streams with an object of class `basic_string`.

The class `basic_stringstream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also
Template class `basic_istringstream`
Class `basic_ostringstream`

### 19.5.1 `basic_stringstream` Constructor

The `basic_stringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

```cpp
explicit basic_stringstream (ios_base::openmode which = ios_base::out | ios_base::out);
explicit basic_stringstream (const basic_string<charT> &str, ios_base::openmode which = ios_base::out | ios_base::out);
```

Remarks

The `basic_stringstream` constructor is overloaded to accept an object of class `basic_string` for input or output.

See Also

`basic_ostringstream`, `basic_istringstream`

Listing: Example of `basic_stringstream::basic_stringstream()` usage

```cpp
#include <iostream>
#include <sstream>
char buf[50] = "ABCD 22 33.33";
char words[50];

int main()
{
    using namespace std;
    stringstream iost;
    char word[20];
    long num;
    double real;
    iost << buf;
    iost >> word;
    iost >> num;
    iost >> real;
```
cout << word << " "
<< num << " "
<< real << endl;
return 0;
}

Result
ABCD 22 33.33

19.5.2 Member functions
The class basic_stringstream has two member functions.

19.5.2.1 basic_stringstream::rdbuf

To retrieve a pointer to the stream buffer.

basic_stringbuf<charT, traits>* rdbuf() const;

Remarks
To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function rdbuf() allows you to retrieve this pointer.

A pointer to an object of type basic_stringbuf sb is returned by the rdbuf function.

See Also
Template class basic_istringstream
Class basic_ostringstream
Listing: Example of basic_stringstream::rdbuf() usage

#include <iostream>
#include <iostream>
#include <sstream>

std::string buf = "CodeWarrior - "Software at Work"";
char words[50];

int main()
{
using namespace std;
    stringstream ist(buf, ios::in);
    istream in(ist.rdbuf());
in.seekg(25);
    in.get(words,50);
    cout << words;
}
return 0;
}

Result
"Software at Work"

19.5.2.2 basic_stringstream::str

To return or assign the basic_string object stored in the buffer.

    basic_string<charT> str() const;
    void str(const basic_string<charT> &s);

Remarks

The function str() freezes the buffer then returns a basic_string object.
The function str(const string s) assigns the value of the string `s' to the stringbuf object.
The no argument version returns a basic_string if successful. The function with an argument has no return.

See Also

basic_stringbuf::str()

basic_ostringstream.str()

basic_istringstream::str()

Listing: Example of basic_stringstream::str() usage

#include <iostream>
#include <sstream>

std::string buf = "CodeWarrior - "Software at Work"\";
char words[50];

int main()
{
    using namespace std;
    stringstream iost(buf, ios::in);
    cout << iost.str();
    return 0;
}

Result
CodeWarrior - "Software at Work"
# Chapter 20
## File Based Streams

Association of stream buffers with files for file reading and writing.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

- Header \texttt{fstream}
- File Streams Type Defines
- Template class \texttt{basic\_filebuf}
- Template class \texttt{basic\_ifstream}
- Template class \texttt{basic\_ofstream}
- Template class \texttt{basic\_fstream}

## 20.1 Header \texttt{fstream}

The header \texttt{<fstream>} defines template classes and types to assist in reading and writing of files.

## 20.2 File Streams Type Defines

A \texttt{FILE} refers to the type \texttt{FILE} as defined in the Standard C Library and provides an external input or output stream with the underlying type of \texttt{char} or \texttt{byte}. A stream is a sequence of \texttt{char} or \texttt{bytes}.

```cpp
typedef basic_filebuf<char> filebuf;
typedef basic_filebuf<wchar_t> wfilebuf;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
```
20.3 Template class basic_filebuf

A class to provide for input and output file stream buffering mechanisms.

The filebuf class is derived from the streambuf class and provides a buffer for file output and or input.

20.3.1 basic_filebuf Constructors

This section describes basic_filebuf constructors.

20.3.1.1 Constructor

To construct and initialize a filebuf object.

basic_filebuf()

Remarks

The constructor opens a basic_filebuf object and initializes it with basic_streambuf<traits>() and if successful is_open() is false.

Listing: For example of basic_filebuf::basic_filebuf() usage:

```cpp
#include <iostream>
#include <fstream>
#include <cstdio>
#include <cstring>

char inFile[] = "ewl-test";

int main()
{
    using namespace std;
    FILE *fp = fopen( inFile, "a+");
    filebuf in(fp);
```
if (!in.is_open())
    { cout << "could not open file"; exit(1); }

char str[] = "\n\ttrademark";
in.sputn(str, strlen(str));
in.close();
return 0;
}

Result:

The file ewl-test now contains:
CodeWarrior "Software at Work"
   trademark

20.3.1.2 Destructor

To remove the basic_filebuf object from memory.

   virtual ~basic_filebuf();

20.3.2 Member functions

The class basic_filebuf provides several functions for file buffer manipulations.

20.3.2.1 basic_filebuf::is_open

Test to ensure filebuf stream is open for reading or writing.

   bool is_open() const

   Remarks

   Use the function is_open() for a filebuf stream to ensure it is open before attempting to do any input or output operation on the stream.

   Returns true if stream is available and open.

   See Also

   For example of basic_filebuf::is_open() usage see basic_filebuf::basic_filebuf
20.3.2.2 basic_filebuf::open

Open a basic_filebuf object and associate it with a file.

```cpp
basic_filebuf<charT, traits>* open
  (const char* c, ios_base::openmode mode);
```

Remarks

You would use the function open() to open a filebuf object and associate it with a file. You may use open() to reopen a buffer and associate it if the object was closed but not destroyed.

If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function is_open() to check for file openings.

If successful the this pointer is returned, if is_open() equals true then a null pointer is returned.

### Table 20-1. Legal basic_filebuf file opening modes

<table>
<thead>
<tr>
<th>Opening Modes</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Only</td>
<td></td>
</tr>
<tr>
<td>ios:: in</td>
<td>&quot;r&quot;</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>ios:: in</td>
</tr>
<tr>
<td>Output Only</td>
<td></td>
</tr>
<tr>
<td>ios:: out</td>
<td>&quot;w&quot;</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>ios:: out</td>
</tr>
<tr>
<td>ios:: out</td>
<td>ios:: trunc</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>ios:: out</td>
</tr>
<tr>
<td>ios:: out</td>
<td>ios:: app</td>
</tr>
<tr>
<td>Input and Output</td>
<td></td>
</tr>
<tr>
<td>ios:: in</td>
<td>ios:: out</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>ios:: in</td>
</tr>
<tr>
<td>ios:: in</td>
<td>ios:: out</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>ios:: in</td>
</tr>
<tr>
<td>ios::binary</td>
<td>ios:: out</td>
</tr>
</tbody>
</table>

Listing: Example of filebuf::open() usage:

```cpp
// The file ewl-test before operation contained:
// CodeWarrior "Software at Work"
#include <fstream>
#include <cstdlib>
char inFile[] = "ewl-test";
```
int main()
{
    using namespace std;
    filebuf in;
    in.open(inFile, ios::out | ios::app);

    if(!in.is_open())
        {cout << "could not open file"; exit(1);}

    char str[] = "\n\tregistered trademark";
    in.sputn(str, strlen(str));
    in.close();
    return 0;
}

Result:

The file ewl-test now contains:
CodeWarrior "Software at Work"
registered trademark

20.3.2.3 basic_filebuf::close

To close a filebuf stream without destroying it.

basic_filebuf<

Remarks

The function close() would remove the stream from memory but will not remove the
filebuf object. You may re-open a filebuf stream that was closed using the close() function.

The this pointer is returned with success, otherwise a null pointer is returned.

See Also

For example of basic_filebuf::close() usage see basic_filebuf::open().

20.3.3 Overridden virtual functions

This section describes overridden virtual functions.

20.3.3.1 basic_filebuf::showmanyc

Overrides basic_streambuf::showmanyc().
virtual int showmanyc();

Remarks
Behaves the same as basic_streambuf::showmanyc.

20.3.3.2 basic_filebuf::underflow

Overrides basic_streambuf::underflow();

virtual int_type underflow();

Remarks
A sequence of characters is read from the input sequence as though they were read from
the associated file into an internal buffer. This must be done so that the class can recover
the position corresponding to each character between intern_buf and intern_end.

20.3.3.3 basic_filebuf::pbackfail

Overrides basic_streambuf::pbackfail().

virtual int_type pbackfail
  (int_type c = traits::eof());

Remarks
This function puts back the characters designated by \( c \) to the input sequence if possible.

Returns \( \text{traits::eof() \ if \ failure} \) and returns either the character put back or
\( \text{traits::not_eof(c) \ for \ success} \).

20.3.3.4 basic_filebuf::overflow

Overrides basic_streambuf::overflow()

virtual int_type overflow
  (int_type c = traits::eof());
Remarks
Behaves the same as basic_streambuf<charT, traits>::overflow(c) except the behavior of consuming characters is performed by conversion.
Returns traits::eof() with failure.

20.3.3.5 basic_filebuf::seekoff

Overrides basic_streambuf::seekoff()

virtual pos_type seekoff
  (off_type off,
   ios_base::seekdir way,
   ios_base::openmode which = ios_base::in | ios_base::out);

Remarks
Sets the offset position of the stream as if using the C standard library function fseek(file, off, whence).
Seekoff function returns a newly formed pos_type object which contains all information needed to determine the current position if successful. Returns an invalid stream position if it fails.

20.3.3.6 basic_filebuf::seekpos

Overrides basic_streambuf::seekpos()

virtual pos_type seekpos
  (pos_type sp,
   ios_base::openmode which =
   ios_base::in | ios_base::out);

Remarks
Seekpos function returns a newly formed pos_type object which contains all information needed to determine the current position if successful. Returns an invalid stream position if it fails.

20.3.3.7 basic_filebuf::setbuf

Overrides basic_streambuf::setbuf()
virtual basic_streambuf<charT traits>* setbuf (char_type* s, streamsize n);

**Remarks**

Setbuf returns zero if the file pointer fp is a null pointer. Otherwise, it calls setvbuf(fp, (char *)buffer, _IOFBF, n * sizeof (E)) to offer the array of n elements beginning at s as a buffer for the stream. If that function returns a nonzero value, the function returns a null pointer. Otherwise, the this pointer is returned to signal success.

### 20.3.3.8 basic_filebuf::sync

Overrides basic_streambuf::sync

    virtual int sync();

**Remarks**

This protected member function returns zero if the file pointer fp is a null pointer. Otherwise, it returns fflush(fp) to flush any pending output to the stream.

### 20.3.3.9 basic_filebuf::imbue

Overrides basic_streambuf::imbue

    virtual void imbue(const locale& loc);

**Remarks**

After this function is called, inserted or extracted characters will be converted according to loc until another call is made to imbue.

### 20.4 Template class basic_ifstream

A class to provide for input file stream mechanisms.
20.4.1 basic_ifstream Constructor

Creates a file stream for input.

```cpp
basic_ifstream();
explicit basic_ifstream
    (const char *s, ios_base::openmode mode = ios_base::in);
```

Remarks

The constructor creates a stream for file input. It is overloaded to either create and initialize when called or to simply create a class and be opened using the `open()` member function. The default opening mode is `ios::in`. See `basic_filebuf::open()` for valid open mode settings.

See `basic_ifstream::open()` for legal opening modes.

See Also

`basic_ifstream::open()` for overloaded form usage.

Listing: Example of basic_ifstream::basic_ifstream() constructor usage:

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>
char inFile[] = "ewl-test";
int main()
{
    using namespace std;
    ifstream in(inFile, ios::in);
    if(!in.is_open())
        {cout << "can't open input file"; exit(1);} 

    char c ='\0';
    while(in.good())
    {
        if(c) cout << c;
        in.get(c);
    }
    in.close();
    return 0;
}
```

Result:

```
CodeWarrior "Software at Work"
```
20.4.2 Member functions

The `ifstream` class has several public member functions for stream manipulations.

20.4.2.1 `basic_ifstream::rdbuf`

The `rdbuf()` function retrieves a pointer to a `filebuf` type buffer.

```
basic_filebuf<charT, traits>* rdbuf() const;
```

Remarks

In order to manipulate for random access or use an `ifstream` stream for both input and output you need to manipulate the base buffer. The function `rdbuf()` returns a pointer to this buffer for manipulation.

Returns a pointer to type `basic_filebuf`.

Listing: Example of `basic_ifstream::rdbuf()` usage:

```
// The ewl-test file contains originally
// CodeWarrior "Software at Work"
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "ewl-test";

int main()
{
    using namespace std;
    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);} 

    ostream Out(inOut.rdbuf());
    char str[] = "\n\tRegistered Trademark";
    inOut.rdbuf()->pubseekoff(0, ios::end);
    Out << str;
    inOut.close();

    return 0;
}
```

Result:

The File now reads:
CodeWarrior "Software at Work"
Registered Trademark

20.4.2.2 `basic_ifstream::is_open`
Test for open stream.

    bool is_open() const

Remarks
Use `is_open()` to test that a stream is indeed open and ready for input from the file.
Returns true if file is open.

See Also
For example of `basic_ifstream::is_open()` usage see `basic_ifstream::basic_ifstream()`

20.4.2.3  `basic_ifstream::open`

Open is used to open a file or reopen a file after closing it.

    void open(const char* s, ios::openmode mode = ios::in);

Remarks
The default open mode is `ios::in`, but can be one of several modes. (see below) A stream is opened and prepared for input or output as selected.

There is no return.

If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

### Table 20-2. Legal `basic_ifstream` file opening modes

<table>
<thead>
<tr>
<th>Opening Modes</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Only</td>
<td></td>
</tr>
<tr>
<td><code>ios::in</code></td>
<td>&quot;r&quot;</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::in`</td>
</tr>
<tr>
<td>Input and Output</td>
<td></td>
</tr>
<tr>
<td>`ios::in</td>
<td>ios::out`</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::in</td>
</tr>
<tr>
<td>`ios::in</td>
<td>ios::out</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::in</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::out</td>
</tr>
</tbody>
</table>

Listing: Example of `basic_ifstream::open()` usage:
```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "ewl-test";

int main()
{
    using namespace std;
    ifstream in;
in.open(inFile);
    if(!in.is_open())
    {
        cout << "can't open input file"; exit(1);
    }
    char c = NULL;
    while((c = in.get()) != EOF)
    {
        cout << c;
    }
    in.close();
    return 0;
}

Result:
CodeWarrior "Software at Work"

20.4.2.4 basic_ifstream::close

Closes the file stream.

    void close();

Remarks

The close() function closes the stream for operation but does not destroy the ifstream object so it may be re-opened at a later time. If the function fails, it calls setstate(failbit), which may throw an exception.

There is no return.

See Also

For example of basic_ifstream::close() usage see basic_ifstream::basic_ifstream()

20.5 Template class basic_ofstream
```
A class to provide for output file stream mechanisms.

**NOTE**

The `basic_ofstream` class supports writing to a file. It uses a `basic_filebuf` object to control the sequence. That object is represented here as `basic_filebuf sb`.

The `basic_ofstream` class provides for mechanisms specific to output file streams.

### 20.5.1 basic_ofstream Constructors

To create a file stream object for output.

```cpp
basic_ofstream();
explicit basic_ofstream
(const char *s, ios_base::openmode mode = ios_base::out | ios_base::trunc);
```

**Remarks**

The class `basic_ofstream` creates an object for handling file output. It may be opened later using the `ofstream::open()` member function. It may also be associated with a file when the object is declared. The default open mode is `ios::out`.

There are only certain valid file opening modes for an ofstream object. See `basic_ofstream::open` for a list of valid opening modes.

**Listing: Example of basic_ofstream::ofstream() usage:**

```cpp
// Before the operation the file ewl-test
// may or may not exist.
#include <iostream>
#include <fstream>
#include <cstdlib>
char outFile[] = "ewl-test";
int main()
{
    using namespace std;
    ofstream out(outFile);
    if(!out.is_open())
        {cout << "file not opened"; exit(1);}
    out << "This is an annotated reference that 
    contains a description\n    of the Working ANSI C++ Standard 
    Library and other\nfacilities of 
    the Embedded Warrior Library. ";
    out.close();
```
template class basic_ofstream

   return 0;

}

Result:

This is an annotated reference that contains a description of the Working ANSI C++ Standard Library and other facilities of the Embedded Warrior Library.

20.5.2 Member functions

The ofstream class provides public member functions for output stream manipulation.

20.5.2.1 basic_ofstream::rdbuf

To retrieve a pointer to the stream buffer.

basic_filebuf<charT, traits>* rdbuf() const;

Remarks

In order to manipulate a stream for random access or other operations you must use the streams base buffer. The member function rdbuf() is used to return a pointer to this buffer.

A pointer to basic_filebuf type is returned.

Listing: Example of basic_ofstream::rdbuf() usage:

```cpp
// The file ewl-test before the operation contains:
// This is an annotated reference that contains a description
// of the Working ANSI C++ Standard Library and other
// facilities of the Embedded Warrior Library

#include <iostream>
#include <fstream>
#include <cstdlib>

char outFile[] = "ewl-test";

int main()
{
    using namespace std;
    ofstream out(outFile, ios::in | ios::out);
    if(!out.is_open())
    {
        cout << "could not open file for output"; exit(1);
    }
    istream inOut(out.rdbuf());
    char ch;
    while((ch = inOut.get()) != EOF)
    {
        cout.put(ch);
    }
    out << "\nAnd so it goes...";
```
```c
out.close();
return 0;
```

**Result:**

This is an annotated reference that contains a description of the Working ANSI C++ Standard Library and other facilities of the Embedded Warrior Library.

And so it goes...

### 20.5.2.2 basic_ofstream::is_open

To test whether the file was opened.

```c
bool is_open();
```

**Remarks**

The `is_open()` function is used to check that a file stream was indeed opened and ready for output. You should always test with this function after using the constructor or the `open()` function to open a stream.

If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

Returns `true` if file stream is open and available for output.

**See Also**

For example of `basic_ofstream::is_open()` usage see `basic_ofstream::ofstream()`

### 20.5.2.3 basic_ofstream::open

To open or re-open a file stream for output.

```c
void open(const char* s, ios_base::openmode mode = ios_base::out);
```

**Remarks**
The function `open()` opens a file stream for output. The default mode is `ios::out`, but may be any valid open mode (see below.) If failure occurs `open()` calls `setstate(failbit)` which may throw an exception.

There is no return.

**Table 20-3. Legal basic_ofstream file opening modes.**

<table>
<thead>
<tr>
<th>Opening Modes</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output only</td>
<td></td>
</tr>
<tr>
<td><code>ios::out</code></td>
<td>&quot;w&quot;</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::out`</td>
</tr>
<tr>
<td>`ios::out</td>
<td>ios::trunc`</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::out</td>
</tr>
<tr>
<td>`ios::out</td>
<td>ios::app`</td>
</tr>
<tr>
<td>Input and Output</td>
<td></td>
</tr>
<tr>
<td>`ios::in</td>
<td>ios::out`</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::in</td>
</tr>
<tr>
<td>`ios::in</td>
<td>ios::out</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::in</td>
</tr>
<tr>
<td>`ios::binary</td>
<td>ios::out</td>
</tr>
</tbody>
</table>

**Listing: Example of basic_ofstream::open() usage:**

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

char outFile[] = "ewl-test";
int main()
{
using namespace std;

ofstream out;
out.open(outFile, ios::out | ios::app);
if(!out.is_open())
{
    cout << "file not opened"; exit(1);
}

out << "\nThis is an annotated reference that "
    << "contains a description\n"
    << "of the Working ANSI C++ Standard "
    << "Library and other\nfacilities of "
    << "the Embedded Warrior Library."
;

out.close();
return 0;
}
```

**Result:**

After the operation ewl-test contained

**Chapter One**
This is an annotated reference that contains a description of the Working ANSI C++ Standard Library and other facilities of the Embedded Warrior Library.

20.5.2.4 basic_ofstream::close

The member function closes the stream but does not destroy it.

```cpp
void close();
```

Remarks

Use the function close() to close a stream. It may be re-opened at a later time using the member function open(). If failure occurs open() calls setstate(failbit) which may throw an exception.

There is no return.

See Also

For example of basic_ofstream::close() usage see basic_ofstream().

20.6 Template class basic_fstream

template class for the association of a file for input and output.

A

20.6.1 basic_fstream Constructor

To construct an object of basic_ifstream for input and output operations.

```cpp
basic_fstream();

explicit basic_fstream (const char *s, ios_base::openmode = ios_base::in | ios_base::out);
```

Remarks

The basic_fstream class is derived from basic_iostream and a basic_filebuf object is initialized at construction.
Listing: Example of basic_fstream:: basic_fstream() usage

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "ewl-test";

int main()
{
    using namespace std;
    fstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);}
    char str[] = "\n\tRegistered Trademark";
    char ch;
    while((ch = inOut.get())!= EOF)
    {
        cout << ch;
    }
    inOut.clear();
    inOut << str;
    inOut.close();
    return 0;
}
```

Result:

CodeWarrior "Software at Work"

The File now reads:

CodeWarrior "Software at Work"
Registered Trademark

20.6.2 Member Functions

The fstream class provides public member functions for input and output stream manipulations.

20.6.2.1 basic_fstream::rdbuf

The rdbuf() function retrieves a pointer to a filebuf type buffer.

```cpp
basic_filebuf<charT, traits>* rdbuf() const;
```
Remarks

In order to manipulate for random access or use an `fstream` stream you may need to manipulate the base buffer. The function `rdbuf()` returns a pointer to this buffer for manipulation.

A pointer to type `basic_filebuf` is returned.

**Listing: Example of basic_fstream::rdbuf() usage**

```c++
// The ewl-test file contains originally
// CodeWarrior "Software at Work"
#include <iostream>
#include <fstream>
#include <cstdlib>
char inFile[] = "ewl-test";
int main()
{
    using namespace std;
    fstream inOut;
inOut.open(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);}
    char str[] = "\n\tRegistered Trademark";
inOut.rdbuf()->pubseekoff(0,ios::end);
inOut << str;
inOut.close();
    return 0;
}
```

Result:
The File now reads:

CodeWarrior "Software at Work"

Registered Trademark

**20.6.2.2 basic_fstream::is_open**

Test to ensure `basic_fstream` file is open and available for reading or writing.

```c++
bool is_open() const
```

Remarks

Use the function `is_open()` for a `basic_fstream` file to ensure it is open before attempting to do any input or output operation on a file.
Returns true if a file is available and open.

See Also

For an example, see Example of basic_fstream::basic_fstream() usage.

### 20.6.2.3 basic_fstream::open

To open or re-open a file stream for input or output.

```cpp
void open (const char* s, ios_base::openmode = ios_base::in | ios_base::out);
```

**Remarks**

You would use the function `open()` to open a `basic_fstream` object and associate it with a file. You may use `open()` to reopen a file and associate it if the object was closed but not destroyed.

If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

There is no return value.

#### Table 20-4. Legal file opening modes

<table>
<thead>
<tr>
<th>Opening Modes</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Only</strong></td>
<td></td>
</tr>
<tr>
<td>ios:: in</td>
<td>&quot;r&quot;</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>&quot;rb&quot;</td>
</tr>
<tr>
<td><strong>Output only</strong></td>
<td></td>
</tr>
<tr>
<td>ios:: out</td>
<td>&quot;w&quot;</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>ios:: out</td>
<td>&quot;w&quot;</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>ios:: out</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td><strong>Input and Output</strong></td>
<td></td>
</tr>
<tr>
<td>ios:: in</td>
<td>&quot;r+&quot;</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>&quot;r+b&quot;</td>
</tr>
<tr>
<td>ios:: in</td>
<td>&quot;w+&quot;</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>&quot;w+b&quot;</td>
</tr>
<tr>
<td>ios:: binary</td>
<td>&quot;ab&quot;</td>
</tr>
</tbody>
</table>

See Also
For an example, see Example of basic_fstream::rdbuf() usage.

20.6.2.4 basic_fstream::close

The member function closes the stream but does not destroy it.

    void close();

Remarks
Use the function close() to close a stream. It may be re-opened at a later time using the member function open(). If failure occurs open() calls setstate(failbit) which may throw an exception.

There is no return value.

See Also
For an example, see Example of basic_fstream::basic_fstream() usage.
Chapter 21
C Library Files

The header `<cstdio>` contains the C++ implementation of the Standard C Headers.

This chapter is constructed in the following sub sections and uses the ISO (International Organization for Standardization) C++ Standard as a guide:

<table>
<thead>
<tr>
<th>Macros</th>
<th>BufSiz</th>
<th>EOF</th>
<th>Filename_MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen_max</td>
<td>L_tmpnam</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>fseek_cur</td>
<td>SEEK_END</td>
<td>SEEK_SET</td>
<td></td>
</tr>
<tr>
<td>stderr</td>
<td>stdin</td>
<td>stdout</td>
<td></td>
</tr>
<tr>
<td>tmp_max</td>
<td>_IoffBF</td>
<td>_IOFBF</td>
<td></td>
</tr>
<tr>
<td>_ionbf</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 21-2. `<cstdio>` Types

| Types: | file_t | size_t |

Table 21-3. `<cstdio>` Functions

<table>
<thead>
<tr>
<th>Functions:</th>
<th>clearerr</th>
<th>fclose</th>
<th>feof</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>fclose</td>
<td>feof</td>
<td></td>
</tr>
<tr>
<td>fgetpos</td>
<td>fflush</td>
<td>fgetc</td>
<td></td>
</tr>
<tr>
<td>fprintf</td>
<td>fputc</td>
<td>fputs</td>
<td></td>
</tr>
<tr>
<td>fread</td>
<td>freopen</td>
<td>fscanf</td>
<td></td>
</tr>
<tr>
<td>fseek</td>
<td>fsetpos</td>
<td>ftell</td>
<td></td>
</tr>
<tr>
<td>fwrite</td>
<td>getc</td>
<td>getchar</td>
<td></td>
</tr>
<tr>
<td>gets</td>
<td>perror</td>
<td>printf</td>
<td></td>
</tr>
<tr>
<td>putc</td>
<td>putchar</td>
<td>puts</td>
<td></td>
</tr>
</tbody>
</table>

Table continues on the next page...
<table>
<thead>
<tr>
<th>Functions:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>remove</td>
<td>rename</td>
<td>rewind</td>
</tr>
<tr>
<td>scanf</td>
<td>setbuf</td>
<td>setvbuf</td>
</tr>
<tr>
<td>sprintf</td>
<td>scanf</td>
<td>tmpnam</td>
</tr>
<tr>
<td>ungetc</td>
<td>vprintf</td>
<td>vfprintf</td>
</tr>
<tr>
<td>vsprintf</td>
<td>tmpfile</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 22
Strstream

The header <strstream> defines streambuf derived classes that allow for the formatting and storage of character array based buffers, as well as their input and output.

The chapter is constructed in the following sub sections and is guided by annex D of the ISO (International Organization for Standardization) C++ Standard.

- Strstreambuf Class a base class for strstream classes
- Istrstream Class a strstream class for input
- Ostrstream Class a strstream class for output
- Strstream Class a class for input and output

22.1 Header strstream

The include file strstream includes three classes for in memory character array based stream input and output.

22.2 Strstreambuf Class

The class strstreambuf is derived from streambuf to associate a stream with an in memory character array.

The strstreambuf class includes virtual protected and public member functions

- freeze freezes the buffer
- pcount determines the buffer size
- str returns a string
- setbuf a virtual function to set the buffer
• `seekoff` a virtual function for stream offset
• `seekpos` a virtual function for stream position
• `underflow` a virtual function for input error
• `pbackfail` a virtual function for put back error
• `overflow` a virtual function for output error

**NOTE**

The template class `streambuf` is an abstract class for deriving various stream buffers whose objects control input and output sequences.

## 22.2.1 Strstreambuf constructors and Destructors

Special constructors and destructors are included for the `strstreambuf` class.

### 22.2.1.1 Constructors

Constructs an object of type `streambuf`.

```cpp
explicit strstreambuf(streamsize alsize_arg = 0);
strstreambuf(void* (*palloc_arg)(size_t),
void (*pfree_arg)(void*));
```

**Dynamic constructors**

```cpp
strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
strstreambuf(const char* gnext_arg, streamsize n);
strstreambuf(signed char* gnext_arg, streamsize n, signed char* pbeg_arg = 0);
strstreambuf(const signed char* gnext_arg, streamsize n);
strstreambuf(unsigned char* gnext_arg, streamsize n, unsigned char* pbeg_arg = 0);
strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

**Remarks**

The constructor sets all pointer member objects to null pointers.
The \texttt{strstreambuf} object is used usually for an intermediate storage object for input and output. The overloaded constructor that is used determines the attributes of the array object when it is created. These might be allocated, or dynamic and are stored in a bitmask type. The first two constructors listed allow for dynamic allocation. The constructors with character array arguments will use that character array for a buffer.

### 22.2.1.2 Destructor

To destroy a \texttt{strstreambuf} object.

```cpp
virtual ~strstreambuf();
```

**Remarks**

Removes the object from memory.

### 22.2.2 \texttt{strstreambuf} Public Member Functions

The public member functions allow access to member functions from derived classes.

#### 22.2.2.1 freeze

To freeze the allocation of \texttt{strstreambuf}.

```cpp
void freeze(bool freeze1 = true);
```

**Remarks**

The function \texttt{freeze()} stops allocation if the \texttt{strstreambuf} object is using dynamic allocation and prevents the destructor from freeing the allocation.

The function \texttt{freeze(false)} releases the freeze to allow for destruction.

There is no return.

**Listing: Example of \texttt{strstreambuf::freeze()} usage:**

```cpp
#include <iostream>
#include <strstream>
#include <string.h>

const int size = 100;
```
int main()
{
    // dynamic allocation minimum allocation 100
    strstreambuf strbuf(size);

    // add a string and get size
    strbuf.sputn( "CodeWarrior", strlen("CodeWarrior"));
    cout << "The size of the stream is: "
        << strbuf.pcount() << endl;
    strbuf.sputc(\0);    // null terminate for output

    // now freeze for no more growth
    strbuf.freeze();

    // try to add more
    strbuf.sputn( " -- Software at Work --", strlen(" -- Software at Work --"));
    cout << "The size of the stream is: "
        << strbuf.pcount() << endl;
    cout << "The buffer contains:\n"
        << strbuf.str() << endl;
    return 0;
}

\[22.2.2.2\] pcount

To determine the effective length of the buffer.

    int pcount() const;

Remarks

The function pcount() is used to determine the offset of the next character position from the beginning of the buffer.

For an example of strstreambuf::pcount() usage refer to strstreambuf::freeze.

\[22.2.2.3\] str

To return the character array stored in the buffer.

    char* str();

Remarks

The function str() freezes the buffer and appends a null character then returns the beginning pointer for the input sequence. The user is responsible for destruction of any dynamically allocated buffer.
**Listing: Example of strstreambuf::str() usage**

```cpp
#include <iostream>
#include <strstream>

const int size = 100;
char buf[size];
char arr[size] = "CodeWarrior - Software at Work";

int main()
{
    ostrstream ostr(buf, size);
    ostr << arr;

    // associate buffer
    strstreambuf *strbuf(ostr.rdbuf());

    // do some manipulations
    strbuf->pubseekoff(10,ios::beg);
    strbuf->sputc('0');
    strbuf->pubseekoff(0, ios::beg);

    cout << "The original array was\n" << arr << "\n\n";
    cout << "The strstreambuf array is\n" << strbuf->str() << "\n\n";
    cout << "The ostrstream array is now\n" << buf;
    return 0;
}
```

### 22.2.3 Protected Virtual Member Functions

Protected member functions that are overridden for stream buffer manipulations by the `strstream` class and derived classes from it.

#### 22.2.3.1 setbuf

To set a buffer for stream input and output sequences.

```cpp
virtual streambuf* setbuf(char* s, streamsize n);
```

**Remarks**

The function `setbuf()` is overridden in `strstream` classes.

The `this` pointer is returned.

#### 22.2.3.2 seekoff
Alters the stream position within one of the controlled sequences if possible.

```cpp
virtual pos_type seekoff(
    off_type off,
    ios_base::seekdir way,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

**Remarks**

The function `seekoff()` is overridden in `sstream` classes.

Returns new stream position if successful. Otherwise, it returns `pos_type(off_type(-1))`.

### 22.2.3.3 seekpos

To alter an input or output stream position.

```cpp
virtual pos_type seekpos(
    pos_type sp,
    ios_base::openmode
    which = ios_base::in | ios_base::out);
```

**Remarks**

The function `seekpos()` is overridden in `sstream` classes.

Returns new stream position if successful. Otherwise, it returns `pos_type(off_type(-1))`.

### 22.2.3.4 underflow

This function attempts to extract the current item from the input buffer and advance the current stream position. The item is returned as `(int)(unsigned char)`.

```cpp
virtual int_type underflow();
```

**Remarks**

The virtual function `underflow()` is called when a character is not available for input.

There are many constraints for `underflow()`.

The pending sequence of characters is a concatenation of end pointer minus the get pointer plus some sequence of characters to be read from input.

Returns the result character if the sequence is not empty, which is the first character in the sequence or the next character in the sequence.
Returns the backup sequence if the beginning pointer is null or the sequence is empty. Otherwise the sequence is the get pointer minus the beginning pointer.

If the position is null, returns traits::eof() to indicate failure. Otherwise, it returns the current element in the input stream.

### 22.2.3.5 pbackfail

To show a failure in a put back operation.

```cpp
virtual int_type pbackfail(int_type c = EOF);
```

**Remarks**

The resulting conditions are the same as the function underflow().

The function pbackfail() is only called when a put back operation really has failed and returns traits::eof. If successful, returns traits::not_eof(c).

### 22.2.3.6 overflow

Consumes the pending characters of an output sequence.

```cpp
virtual int_type overflow (int_type c = EOF);
```

**Remarks**

The pending sequence is defined as the concatenation of the put pointer minus the beginning pointer plus either the sequence of characters or an empty sequence, unless the beginning pointer is null in which case the pending sequence is an empty sequence.

This function is called by sputc() and sputn() when the buffer is not large enough to hold the output sequence.

Overriding this function requires that:

- When overridden by a derived class how characters are consumed must be specified.
- After the overflow either the beginning pointer must be null or the beginning and put pointer must both be set to the same non-null value.
The function may fail if appending characters to an output stream fails or failure to set the previous requirement occurs.

The function returns `traits::eof()` for failure or `traits::not_eof(c)` to indicate success.

### 22.3  **istream Class**

The class `istream` is used to create and associate a stream with an array for input.

The `istream` class includes the following facilities

- **Constructors** and **Destructor** to create and remove an `istream` object
- `rdbuf` to access the buffer
- `str` returns the buffer

### 22.3.1  **Constructors and Destructor**

The `istream` class has an overloaded constructor.

#### 22.3.1.1  **Constructors**

Creates an array based stream for input.

```cpp
explicit istrstream(const char* s);
explicit istrstream(char* s);
istreamstream(const char* s, streamsize n);
istreamstream(char* s, streamsize n);
```

**Remarks**

The `istreamstream` constructor is overloaded to accept a dynamic or pre-allocated character based array for input. It is also overloaded to limit the size of the allocation to prevent accidental overflow.

**Listing: Example of usage.**

```cpp
#include <iostream>
#include <strstream>

char buf[100] ="double 3.21 string array int 321";

int main()
{
    char arr[4][20];
    double d;
    long i;
```
```cpp
istream istr(buf);
cout << arr[0] << " is " << d << "\n"
  << arr[3] << " is " << i << endl;
return 0;
}

Result:
double is 3.21
string is array
int is 321

### 22.3.1.2 Destructor

To destroy an istream object.

```cpp
t
virtual ~istream();

**Remarks**
The `istream` destructor removes the `istream` object from memory.

### 22.3.2 Public Member Functions

There are two public member functions.

#### 22.3.2.1 rdbuf

Returns a pointer to `strstreambuf`.

```cpp
strstreambuf* rdbuf() const;

**Remarks**
To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the stream's buffer. The function `rdbuf()` allows you to retrieve this pointer.

Returns a pointer to `strstreambuf`.

For an example of `istream::rdbuf()` usage refer to `strstreambuf::str()`
22.3.2.2  str

Returns a pointer to the stored array.

char* str();

Remarks

The function str() freezes and terminates the character array stored in the buffer with a null character. It then returns the null terminated character array.

A null terminated char array is returned.

Listing: Example of iostream::str() usage.

```cpp
#include <iostream>
#include <sstream>

const int size = 100;
char buf[size] = "CodeWarrior - Software at Work";

int main()
{
    istringstream istr(buf, size);
    cout << istr.str();
    return 0;
}
```

Result:

```
CodeWarrior - Software at Work
```

22.4  ostrstream Class

The class ostrstream is used to create and associate a stream with an array for output.

22.4.1  Constructors and Destructor

The ostrstream class has an overloaded constructor.

22.4.1.1  Constructors

Creates a stream and associates it with a char array for output.
Remarks

The `ostrstream` class is overloaded for association with a pre allocated array or for dynamic allocation.

When using an ostrstream object the user must supply a null character for termination. When storing a string which is already null terminated that null terminator is stripped off to allow for appending.

Listing: Example of ostrstream usage.

```cpp
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "Ask the teacher anything you want to know" << ends;
    istream inOut(out.rdbuf());
    char c;
    while( inOut.get(c) ) cout.put(c);
    return 0;
}
```

Result:

Ask the teacher anything you want to know

22.4.1.2 Destructor

Destroys an `ostrstream` object.

```cpp
virtual ~ostrstream();
```

Remarks

An ostrstream destructor removes the `ostrstream` object from memory.

22.4.2 Public Member Functions

The `ostrstream` class has four public member functions.
22.4.2.1 freeze

Freezes the dynamic allocation or destruction of a buffer.

If N is nonzero (the default), the string associated with this `ostrstream' should be declared not to change dynamically. While frozen, the string will not be reallocated if it needs more space, and will not be deallocated when the ostrstream is destroyed.

```cpp
void ostrstream::freeze ([int N])
```

Remarks

This member function calls `rdbuf()-> freeze(freezeit)'.

Listing: Example of ostrstream freeze() usage.

```cpp
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "CodeWarrior " << 1234;
    out << "the size of the array so far is " << out.pcount() << " characters \n";
    out << " Software" << '\0';
    out.freeze();       // freezes so no more growth can occur
    out << " at work" << ends;
    out << "the final size of the array is " <<out.pcount() << " characters \n";
    cout << out.str() << endl;
    return 0;
}
```

22.4.2.2 pcount

Determines the number of bytes offset from the current stream position to the beginning of the array.

```cpp
int pcount() const;
```

Remarks

The function `pcount()' is used to determine the offset of the array. This may not equal to the number of characters inserted due to possible positioning operations.

Returns an `int_type` that is the length of the array.

Listing: Example of ostrstream pcount() usage.
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "CodeWarrior " << 1234 << ends;
    out << "the size of the array so far is "
        << out.pcount() << " characters \n";
    out << " Software at work" << ends;
    out << "the final size of the array is "
        << out.pcount() << " characters \n";
    cout << out.str() << endl;
    return 0;
}

22.4.2.3 rdbuf

To retrieve a pointer to the streams buffer.

    stringstream* rdbuf() const;

Remarks

To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function rdbuf() allows you to retrieve this pointer.

Returns a pointer to stringstream.

For an example of ostrstream rdbuf() usage refer to streambuf::pubseekoff()

22.4.2.4 str

Returns a pointer to a character array.

    char* str();

Remarks
The function `str()` freezes any dynamic allocation.
Returns a null terminated character array.
For an example of `ostrstream str()` usage refer to `ostrstream::freeze()

### 22.5 Strstream Class

The class `strstream` associates a stream with an array buffer for input and or output.

#### 22.5.1 Strstream Types

The `strstream` class type defines a `char_type`, `int_type`, `pos_type` and `off_type`, for stream positioning and storage.

#### 22.5.2 Constructors and Destructor

Specialized constructors and destructors are provided.

##### 22.5.2.1 Constructors

Creates a stream and associates it with a character array for input and output.

```cpp
strstream();
strstream(char* s, int n, ios_base::openmode mode =
ios_base::in|ios_base::out);
```

**Remarks**

The `strstream` constructor is overloaded for association with a pre allocated array or for dynamic allocation.

##### 22.5.2.2 Destructor

Destroys a `strstream` object.

```cpp
virtual ~strstream();
```
Remarks
Removes the `strstream` object from memory.

### 22.5.3 Public Member Functions

The class `strstream` has four public member functions.

#### 22.5.3.1 freeze

Freezes the dynamic allocation or destruction of a buffer.

```cpp
void freeze(bool freeze = true);
```

Remarks
The function `freeze` stops dynamic allocation of a buffer.

#### 22.5.3.2 pcount

Determines the number of bytes offset from the current stream position to the beginning of the array.

```cpp
int pcount() const;
```

Remarks
The function `pcount()` is used to determine the offset of the array. This may not equal to the number of characters inserted due to possible positioning operations.

Returns an `int_type` that is the length of the array.

#### 22.5.3.3 rdbuf

Retrieves a pointer to the streams buffer.
`strstreambuf* rdbuf() const;`

**Remarks**

To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer.

Returns a pointer to `strstreambuf`.

### 22.5.3.4 str

Returns a pointer to a character array.

```cpp
char* str();
```

**Remarks**

The function `str()` freezes any dynamic allocation.

Returns a null terminated character array.
Chapter 23
Bitvector Class Library

The bitvector class template interface is based on the std::vector<bool> interface. It is a dynamically sized array of bools packed into 1 bit per bool representation. In the default shipping configuration, Metrowerks::bitvector<> and std::vector<bool> provide identical functionality and performance. However clients can use the statement #define _EWL_NOVECTOR_BOOL which removes the std::vector<bool> specialization, causing std::vector<bool> to behave like any other vector<T>. In this configuration, Metrowerks::bitvector<> remains available and as described herein. This allows clients the opportunity to use both packed and unpacked arrays of bool in the same application.

The bitvector class consists of:

- Nested types
- Constructors
- Capacity
- Iteration
- Access
- Insertion
- Erasure
- Miscellaneous
- Namespace scope functions

Listing: Class bitvector synopsis

namespace Metrowerks {
    template <class Allocator = std::allocator<bool> >
    class bitvector
    {
        public:
            //  types:
            typedef Allocator allocator_type;
            typedef typename allocator_type::size_type size_type;
            typedef typename allocator_type::difference_type difference_type;
            typedef bool value_type;

            class reference;
            class const_reference;
            class pointer;
            class const_pointer;
            class iterator;  // random access
class  const_iterator;  // random access

typedef std::reverse_iterator<iterator>  reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
explicit bitvector(const allocator_type& a = Allocator());
explicit bitvector(size_type n, bool x = false, const allocator_type&
a = Allocator());

template <class InputIterator>
  bitvector(InputIterator first, InputIterator last, const
  allocator_type& a = Allocator());
bitvector(const bitvector& x);
bitvector& operator=(const bitvector& x);
~bitvector();
size_type size() const;
bool      empty() const;
size_type capacity() const;
size_type max_size() const;
void reserve(size_type n);
allocator_type get_allocator() const;
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;
reference       front();
const_reference front() const;
reference       back();
const_reference back() const;
reference       operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference       at(size_type n);
void assign(size_type n, bool x);
template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
void push_back(bool x);
void pop_back();
iterator insert(iterator position, bool x);
void       insert(iterator position, size_type n, bool x);

template <class InputIterator>
  void insert(iterator position, InputIterator first, InputIterator last);
void clear();
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void resize(size_type sz, bool c = false);
void swap(bitvector& x);
void flip();
bool invariants() const;
};

template <class Allocator>
bool operator==(const bitvector<Allocator>& x, const bitvector<Allocator>& y);
bool operator>=(const bitvector<Allocator>& x, const bitvector<Allocator>& y);

template <class Allocator>
bool operator<=(const bitvector<Allocator>& x, const bitvector<Allocator>& y);

template <class Allocator>
void swap(bitvector<Allocator>& x, bitvector<Allocator>& y);

} // Metrowerks

23.1 Nested types

This section describes nested types.

23.1.1 allocator_type

typedef Allocator allocator_type;

The single template parameter of `bitvector` must be an allocator meeting the standard allocator requirements. This parameter defaults to `std::allocator<bool>`. Clients can refer to this type via the nested name: `allocator_type`.

23.1.2 size_type

typedef typename allocator_type::size_type size_type;

`size_type` is constrained to be an unsigned integral type capable of representing all `bitvector` capacities. It is introduced into `bitvector` as a nested type of the allocator. The default type is `std::size_t`.

23.1.3 difference_type

typedef typename allocator_type::difference_type
difference_type;


difference_type is a signed integral type capable of representing the difference between two bitvector iterators. It is introduced into bitvector as a nested type of the allocator. The default type is std::ptrdiff_t.

23.1.4 value_type

typedef bool value_type;

For compatibility with other standard containers, the nested type value_type is defined as bool.

23.1.5 reference

class reference;

The nested class reference is a "smart reference" class which emulates a reference to an internal bool. An actual reference (bool&) can not be used here since the internal bools are stored as a single bit. In most cases the behavior will be identical to bool&. One exception is that the reference has a member function named flip() that will change the value of the underlying bit.

#include <bitvector>
#include <algorithm>
#include <cassert>

int main()
{
    Metrowerks::bitvector<> v(3);
    Metrowerks::bitvector<>::reference r = v[0];
    assert(v[0] == false);
    assert(r == false);
    r = true;
    assert(v[0] == true);
    r.flip();
    assert(v[0] == false);
    v[1] = true;
    swap(r, v[1]);
    assert(r == true);
    assert(v[0] == true);
    assert(v[1] == false);
    Metrowerks::bitvector<>::pointer p = &r;
    assert(*p == true);
    *p = false;
assert(v[0] == false);
assert(r == false);
assert(*p == false);
}

NOTE
swap can be called with this reference type, even with an rvalue reference. As it applies to std::vector<bool>::reference, this is an extension to the standard. Another extension to better emulate a real reference is that you can take the address of a reference that yields the nested type pointer.

23.1.6 const_reference

class const_reference;

The nested class const_reference is a "smart reference" class which emulates a const reference to an internal bool. An actual reference (const bool&) can not be used here since the internal bools are stored as a single bit. In most cases the behavior will be identical to const bool&. As it applies to std::vector<bool>::const_reference, this is an extension to the standard. The standard specifies that std::vector<bool>::const_reference is just a bool. But the following code demonstrates how this proxy class more closely emulates a const bool& than does a bool. Another extension to better emulate a real const reference is that you can take the address of a const_reference which yields the nested type const_pointer.

#include <bitvector>
#include <cassert>

int main()
{
    Metrowerks::bitvector<> v(3);
    Metrowerks::bitvector<>::const_reference cr = v[0];
    assert(cr == false);
    v[0] = true;
    assert(cr == true);
    Metrowerks::bitvector<>::const_pointer cp = &cr;
    assert(*cp == true);
}

23.1.7 iterators and pointers

class      pointer;
class      const_pointer;
class      iterator;
class      const_iterator;
The nested types iterator and pointer are the same type, as are const_iterator and const_pointer. Both are random access iterators, except that they return reference and const_reference respectively when dereferenced (as opposed to bool& and const bool&).

The following standard algorithms are specialized for iterator and const_iterator as appropriate. They will operate on a word at a time instead of a bit at a time for superior performance.

```cpp
Iterator copy(Iterator first, Iterator last, Iterator result);
Iterator copy_backward(Iterator first, Iterator last, Iterator result);
void fill_n(Iterator first, size_type n, const T& value);
void fill(Iterator first, Iterator last, const T& value);
bool equal(Iterator first1, Iterator last1, Iterator first2);
```

### 23.2 Constructors

This section explains Constructors.

```cpp
explicit bitvector(const allocator_type& a = Allocator());
```

Constructs an empty bitvector, with the supplied (or defaulted) allocator. It will not throw an exception unless constructing or copying the allocator throws an exception. The default allocator, std::allocator<bool>, will not throw in this context.

Postcondition: size() == 0 and capacity() == 0. If an allocator was supplied then get_allocator() == a, else get_allocator() == Allocator().

```cpp
explicit bitvector(size_type n, bool x = false, const allocator_type& a = Allocator());
```

Constructs a bitvector of length n with all values set to x.

Postcondition: size() == n and capacity() >= n. All elements are equal to x. If an allocator was supplied then get_allocator() == a, else get_allocator() == Allocator().

```cpp
template <class InputIterator>
bitvector(InputIterator first, InputIterator last, const allocator_type& a = Allocator());
```

Constructs a bitvector from the range [first, last).
Postcondition: \texttt{size()} == \texttt{distance(first, last)} and \texttt{capacity()} >= \texttt{size()}. All elements are equal to the corresponding values in the range \([first, last)\). If an allocator was supplied then \texttt{get_allocator()} == \texttt{a}, else \texttt{get_allocator()} == \texttt{Allocator().}

\begin{verbatim}
bitvector(const bitvector& x);
\end{verbatim}

Constructs a copy of the bitvector \(x\).

Postcondition: \texttt{*this == x}.

\textbf{NOTE}

The capacity of \(x\) is not necessarily duplicated in \texttt{*this}. In general, the copy will be done with the least amount of capacity sufficient to hold \texttt{size()} elements.

### 23.2.1 Destructor

\begin{verbatim}
~bitvector();
\end{verbatim}

Destroys the bitvector and throws nothing.

### 23.2.2 Assignment

\begin{verbatim}
bitvector& operator=(const bitvector& x);
\end{verbatim}

Assigns \(x\) to \texttt{*this}.

Postcondition: \texttt{*this == x}.

If \texttt{capacity()} >= \texttt{x.size()}, no exception can be thrown.

\begin{verbatim}
void assign(size_type n, bool x);
\end{verbatim}

Assigns to \texttt{*this} \(n\) copies of \(x\).

Postcondition: \texttt{*this == bitvector(n, x)}.

If \texttt{capacity()} >= \texttt{n}, no exception can be thrown.
template <class InputIterator>

    void assign(InputIterator first, InputIterator last);

Assigns to *this the range [first, last).
Precondition: first and last are not iterators into this bitvector.
Postcondition: *this == bitvector(first, last).
If capacity() \( \geq \) distance(first, last), and if no operations on the InputIterator type can throw, then no exception can be thrown.

23.3 Capacity

This section describes capacity functions.

23.3.1 size

    size_type size() const;

Returns the current number of elements in the bitvector.
Throws nothing.

23.3.2 empty

    bool empty() const;

Returns size() == 0.
Throws nothing.
23.3.3 capacity

    size_type capacity() const;

Returns the maximum size that can be achieved before a memory allocation is required. Throws nothing.

23.3.4 max_size

    size_type max_size() const;

Returns a maximum size that the bitvector could grow, assuming sufficient memory. This is a design-time limit. Throws nothing.

23.3.5 reserve

    void reserve(size_type n);

If \( n > \text{capacity()} \) will attempt to acquire a capacity() greater to or equal to \( n \), an exception is thrown on failure. The precise type of the exception thrown is dictated by the allocator. The default allocator will throw a \texttt{std::bad_alloc} on failure. If an exception is thrown, there are no effects. If \( n \leq \text{capacity()} \) then there are no effects.

Postcondition: \( \text{capacity()} \geq n \).

23.3.6 get_allocator

    allocator_type get_allocator() const;
Returns a copy of the allocator that the bitvector was constructed with. If the copy constructor of the allocator_type can not throw an exception then get_allocator() is also a non-throwing operation.

23.4 Iteration

This section describes Iteration functions.

```cpp
iterator begin();
const_iterator begin() const;
```

Returns an iterator referring to the first element in the bitvector. If the bitvector is empty, then returns an iterator equal to end().

Throws nothing.

```cpp
iterator end();
const_iterator end() const;
```

Returns an iterator referring to one past the last element in the bitvector. If the bitvector is empty, then returns an iterator equal to begin().

Throws nothing.

```cpp
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

Returns std::reverse_iterator's which operate over the range of the bitvector but in reverse order.

Throws nothing.

23.5 Access

This section describes access functions.

23.5.1 front
reference       front();
const_reference front() const;

Returns a reference to the first element in the `bitvector`.
Precondition: The `bitvector` is not empty.

Throws nothing.

reference       back();
const_reference back() const;

Returns a reference to the last element in the `bitvector`.
Precondition: The `bitvector` is not empty.

Throws nothing.

reference       operator[](size_type n);
const_reference operator[](size_type n) const;

Returns a reference to the \( n \)th element in the `bitvector`.
Precondition: \( n < \text{size}() \).

Throws nothing.

const_reference at(size_type n) const;
reference       at(size_type n);

Returns a reference to the \( n \)th element in the `bitvector`.

Throws nothing if \( n < \text{size}() \), else throws a `std::out_of_range` object. If an exception is thrown, there are no effects.

### 23.6 Insertion

This section defines insertion functions.

#### 23.6.1 `push_back`

```cpp
void push_back(bool x);
```

Appends \( x \) into the `bitvector`.

Postcondition: If an exception is not thrown, \( \text{size}() \) is increased by one and \( \text{back}() == x \).
If an exception is thrown, there are no effects.

### 23.6.2 insert

```cpp
iterator insert(iterator position, bool x);
```

Inserts x into the bitvector at position. All elements in the range [position, end()) are moved down to make room for x. The returned iterator refers to the newly inserted element having value x.

**Precondition:** position is an iterator into this bitvector.

**Postcondition:** If an exception is not thrown, size() is increased by one and *returned_iterator == x.

If an exception is thrown, there are no effects.

```cpp
void insert(iterator position, size_type n, bool x);
```

Inserts n copies of x into the bitvector at position. All elements in the range [position, end()) are moved down to make room for the newly inserted elements.

**Precondition:** position is an iterator into this bitvector.

**Postcondition:** If an exception is not thrown, size() is increased by n. The range [position, position+n) will all have value x.

If an exception is thrown, there are no effects.

```cpp
template <class InputIterator>
void insert(iterator position, InputIterator first, InputIterator last);
```

Inserts the range [first, last) into the bitvector at position. All elements in the range [position, end()) are moved down to make room for the newly inserted elements.

**Precondition:** position is an iterator into this bitvector. first and last are not iterators into this bitvector.

**Postcondition:** If an exception is not thrown, size() is increased by distance(first, last).

If an exception is thrown other than by operations on InputIterator, there are no effects.
23.7 Erasure

This section defines erasure functions.

23.7.1 pop_back

    void pop_back();

Removes the last element in the bitvector.
Precondition: The bitvector is not empty.
Postcondition: size() is decreased by one.
Throws nothing.

23.7.2 clear

    void clear();

Removes all elements in the bitvector.
Postcondition: size() == 0.
Throws nothing.

23.7.3 erase

    iterator erase(iterator position);
Removes the element at position. Elements in the range (position, end) are moved down by one in the bitvector. An iterator pointing to the element just after the erased element, in the modified bitvector, is returned.

Precondition: position is a dereferenceble iterator into this bitvector.

Postcondition: size() is decreased by one.

Throws nothing.

    iterator erase(iterator first, iterator last);

Removes the range of elements [first, last). Elements in the range (last, end) are moved down by distance(first, last) in the bitvector. An iterator pointing to the element just after the erased range, in the modified bitvector, is returned.

Precondition: first is a dereferenceble iterator into this bitvector. last is an iterator into this bitvector. first <= last.

Postcondition: size() is decreased by last-first.

Throws nothing.

23.8 Miscellaneous
This section defines miscellaneous functions.

23.8.1 resize

    void resize(size_type sz, bool c = false);

Changes the size() of the bitvector to sz. If sz is greater than the current size(), extra elements are appended with the value c.

Postcondition: size() == sz.

Throws nothing if sz <= capacity(). If an exception is thrown, there are no effects.
23.8.2 swap

```cpp
void swap(bitvector& x);
```

Swaps the contents of *this and x. If the allocator_type contains state, the allocators are also swapped using an unqualified call to swap, with std::swap in scope.

Postcondition: *this == previous value of x and x == previous value of *this.

If swap on the allocator_type can not throw, then this operation will not throw an exception.

23.8.3 flip

```cpp
void flip();
```

Changes the value of every element.

Throws nothing.

23.8.4 invariants

```cpp
bool invariants() const;
```

This function checks the objects internal invariants and returns true if all are satisfied. If it returns false, it indicates a bug either in the bitvector implementation, or in client code. A common way to use this member is:

```cpp
assert(v.invariants());
```

Throws nothing.

23.9 Namespace scope functions

This section explains Namespace scope functions.
template <class Allocator>
bool
operator==(const bitvector<Allocator>& x, const
bitvector<Allocator>& y);

Returns x.size() == y.size() && std::equal(x.begin(), x.end(), y.begin());

Throws nothing.

template <class Allocator>
bool
operator!=(const bitvector<Allocator>& x, const
bitvector<Allocator>& y);

Returns !(x == y);

Throws nothing.

template <class Allocator>
bool
operator< (const bitvector<Allocator>& x, const
bitvector<Allocator>& y);

Returns std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());

Throws nothing.

template <class Allocator>
bool
operator> (const bitvector<Allocator>& x, const
bitvector<Allocator>& y);

Returns y < x;

Throws nothing.

template <class Allocator>
bool
operator>=(const bitvector<Allocator>& x, const
bitvector<Allocator>& y);

Returns !(x < y);

Throws nothing.

template <class Allocator>
bool
operator<=(const bitvector<Allocator>& x, const
bitvector<Allocator>& y);

Returns !(y < x);

Throws nothing.

template <class Allocator>
void
swap(bitvector<Allocator>& x, bitvector<Allocator>& y);

Calls x.swap(y);
Chapter 24
EWL_Utility

This chapter is a reference guide to the General utility support in the Embedded Warrior Library.

This chapter consists of utilities for support of non standard headers.

- Header ewl_utlity
- Basic Compile-Time Transformations
- Type Query
- CV Query
- Type Classification
- POD classification
- Miscellaneous

24.1 Header ewl_utlity

The purpose of this header is to offer a collection of non-standard utilities collected under the namespace Metrowerks.

These utilities are of a fundamental nature, and are typically used in other utilities, rather than top level code. Example usage assumes that a declaration or directive has been previously issued.

NOTE
This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks. Concepts and ideas co-developed on Boost.

http://www.boost.org/
NOTE
Unnamed namespaces are displayed using a compiler generated unique name that has the form: __unnamed_<filename> where <filename> is the source file name of the main translation unit that contains the unnamed namespace.

NOTE
When generating multiple template instantiations, the compiler may choose to optimize functions that have same binary representation regardless of the specialization being compiled. This results in smaller code, but while debugging no breakpoints can be placed inside optimized functions. To alleviate this, use the "-Xcfe -fl=dont_inline" switch. For more information about dont_inline swith, see the <product> C/C++ Compiler User Guide.

24.2 Basic Compile-Time Transformations
A collection of templated struct types which can be used for simple compile-time transformations of types.

24.2.1 remove_const
Will remove the top level const (if present) from a type.

```cpp
typedef typename remove_const<T>::type non_const_type;
```

Remarks
The resulting "non_const_type" will be the same as the input type T, except that if T is const qualified, that constant qualification will be removed.

Listing: Example of remove Const

```cpp
typedef typename remove_const <const int>::type Int;
Int has type int.
```

24.2.2 remove_volatile
Will remove the top level volatile (if present) from a type.

```cpp
typedef typename
       remove_volatile<T>::type non_volatile_type;
```

**Remarks**

The resulting "non_volatile_type" will be the same as the input type \( T \), except that if \( T \) is volatile qualified, that volatile qualification will be removed.

**Listing: Example of remove_volatile**

```cpp
typedef typename remove_volatile <volatile int>::type Int;
Int has type int.
```

### 24.2.3 remove_cv

Will remove the top level qualifiers (const and/or volatile, if present) from a type.

```cpp
typedef typename
       remove_cv<T>::type non_qualified_type;
```

**Remarks**

The resulting "non_qualified_type" will be the same as the input type \( T \), except that if \( T \) is cv qualified, the qualifiers will be removed.

**Listing: Example of remove_cv**

```cpp
typedef typename remove_cv <const int>::type Int;
Int has type int.
```

### 24.2.4 remove_pointer

If given a pointer, returns the type being pointed to. If given a non-pointer type, simply returns the input.

```cpp
typedef typename
       remove_pointer<T>::type pointed_to_type;
```

**Listing: Example of remove-pointer**

```cpp
typedef typename
       remove_pointer<volatile*const>::type IntPtr;

typedef typename remove_pointer<IntPtr>::type Int;
```
IntPtr will have type type const int*volatile. Int will have the type const int.

24.2.5 remove_reference

If given a reference, returns the type being referenced. If given a non-reference, simply returns the input.

```cpp
typedef typename remove_reference<T>::type referenced_type;
```

Listing: Example of remove_reference

```cpp
typedef typename remove_reference<int&>::type Int;
typedef typename remove_reference<const int&>::type ConstInt;
```

Int has the type int, and ConstInt has the type const int.

24.2.6 remove_bounds

If given an array type, will return the type of an element in the array. If given a non-array type, simply returns the input.

```cpp
typedef typename remove_bounds<T>::type Element;
```

Listing: Example of remove_bounds

```cpp
typedef int IntArray[4];
typedef typename remove_bounds<IntArray>::type Int;
```

Int has the type int.

24.2.7 remove_all

This transformation will recursively remove cv qualifiers, pointers, references and array bounds until the type is a fundamental type, enum, union, class or member pointer.

```cpp
typedef typename remove_all<T>::type fundamental_type;
```

Listing: Example of remove_all
typedef const int** Array[4];
typedef typename remove_all<Array*>::type Int;

Int has the type int.

24.3 Type Query
The following structs perform basic queries on one or more types and return a bool value.

24.3.1 is_same

This struct can be used to tell if two types are the same type or not.

    bool b = is_same<T, U>::value;

Listing: Example of is_same

    bool b = is_same<const int, int>::value;
The resulting value is false. int and const int are two distinct types.

24.4 CV Query

This section defines CV Query functions.

24.4.1 is_const

Returns true if type has a top level const qualifier, else false.

    bool b = is_const<T>::value;

Listing: Example of is_const

    bool b = is_const<const int>::value;
The resulting value is true.
24.4.2 is_volatile

Returns true if type has a top level volatile qualifier, else false.

```c++
bool b = is_volatile<T>::value;
```

Listing: Example of is_volatile

```c++
bool b = is_volatile<const int>::value;
The resulting value is false.
```

24.5 Type Classification

The following structs implement classification as defined by section 3.9 in the C++ standard.

All types can be classified into one of ten basic categories:

- integral
- floating
- void
- pointer
- member pointer
- reference
- array
- enum
- union
- class

Top level cv qualifications do not affect type classification. For example, both const int and int are considered to be of integral type.

```c++
bool b = is_XXX<T>::value;
```

where `is_XXX` is one of the ten basic categories.

- is_integral
- is_floating
- is_void
- is_pointer
- is_member_pointer
- is_reference
There are also five categories that are made up of combinations of the ten basic categories:

- **is_arithmetic** - is_integral or is_floating
- **is_fundamental** - is_arithmetic or is_void
- **is_scalar** - is_arithmetic or is_pointer or is_member_pointer or is_enum
- **is_compound** - not is_fundamental
- **is_object** - anything but a void or reference type

The classifications: **is_enum** and **is_union** do not currently work automatically. Enumerations and unions will be mistakenly classified as class type. This can be corrected on a case by case basis by specializing **is_enum_imp** or **is_union_imp**. These specializations are in the Metrowerks::details namespace.

**is_extension** is also provided for those types that we provide as an extension to the C++ standard. **is_extension<T>::value** will be false for all types except for long long and unsigned long long.

**has_extension** is a modified form of **is_extension** that answers to true if a type is either an extension or contains an extension.

**Listing: Example of is_integral**

```cpp
bool b = is_integral<volatile int>::value;
```

The value of b is true.

**Listing: Example of Metrowerks::details namespace**

```cpp
enum MyEnum {zero, one, two};
```

```cpp
template <>
struct Metrowerks::details::is_enum_imp<MyEnum>
{
    {static const bool value = true;};
```

**Listing: Example of is_extension and has_extension**

```cpp
is_extension<long long*>::value; // false
has_extension<long long*>::value; // true
```

### 24.5.1 **is_signed / is_unsigned**
These structs only work on arithmetic types. The type must be constructable by an int and be less-than comparable.

**Remarks**

In the Example of is_signed and is_unsigned the signedness of char is tested. Since it is implementation-defined whether or not char is signed, this is a way to find out how it is defined. Either b1 will be true and b2 false, or vice-versa.

**Listing: Example of is_signed and is_unsigned**

```cpp
bool b1 = is_signed<char>::value;
bool b2 = is_unsigned<char>::value;
```

### 24.6 POD classification

Four structs classify types as to whether or not they have trivial special members as defined in section 12 of the C++ standard.

- has_trivial_default_ctor
- has_trivial_copy_ctor
- has_trivial_assignment
- has_trivial_dtor

This library will answer correctly for non-class types. But user defined class types will always answer false to any of these queries. If you create a class with trivial special members, and you want that class to be able to take advantage of any optimizations that might arise from the assumption of trivial special members, you can specialize these structs:

Note that in the Example of specialized structs these specializations need not worry about cv qualifications. The higher level has_trivial_XXX structs do that for you.

Finally there is an is_POD struct that will answer true if a type answers true on all four of the above queries.

**Listing: Example of specialized structs**

```cpp
template <>
struct Metrowerks::details::class_has_trivial_default_ctor<MyClass>
{
    static const bool value = true;
};

template <>
struct Metrowerks::details::class_has_trivial_copy_ctor<MyClass>
{
    static const bool value = true;
};
```
template <>
struct Metrowerks::details::class_has_trivial_assignment<MyClass>
   {static const bool value = true;};

template <>
struct Metrowerks::details::class_has_trivial_dtor<MyClass>
   {static const bool value = true;};

## 24.7 Miscellaneous

Miscellaneous utility functions are included in the EWL Utilities library.

### 24.7.1 compile_assert

This is a compile time assert. This is a very basic version of this idea. Can be used to test assertions at compile time.

**Listing: Example of compile_assert use**

```c++
#include <ewl_utility>

template <class T>
T foo(const T& t)
{
   Metrowerks::compile_assert<sizeof(T) >= sizeof(int)>
   T_Must_Be_At_Least_As_Big_As_int;
   //...
   return t;
}

int main()
{
   int i;
   foo(i); // ok
   char c;
   foo(c); // Error : illegal use of incomplete struct/union/class 'Metrowerks::compile_assert<0>'
```

### 24.7.2 array_size

Given an array type, you can get the size of the array with array_size.
The code fragment `array_size<type>::value` will only compile if `type` is an array. It won't compile if `type` is a union, struct or class.

**Listing: Example usage of array_size**

typedef int Array[10];
size_t n = array_size<Array>::value;

`n` has the value of 10.

**24.7.3 can_derive_from**

The code fragment `can_derive_from<T>::value` will be true if `T` is a class (but not a union), otherwise it will be false. Only classes which are not unions can be derived from.

```cpp
bool b = can_derive_from<T>::value;
```

**24.7.4 call_traits**

This struct is a collection of type definitions that ease coding of template classes when the template parameter may be a non-array object, an array, or a reference. The type definitions specify how to pass a type into a function, and how to pass it back out either by value, reference or const reference. The interface is:

```cpp
    call_traits<T>::value_type
    call_traits<T>::reference
    call_traits<T>::const_reference
    call_traits<T>::param_type
```

**Remarks**

The first three types are suggestions on how to return a type from a function by value, reference or const reference. The fourth type is a suggestion on how to pass a type into a method.

The `call_traits` struct is most useful in avoiding references to a reference which are currently illegal in C++. Another use is in helping to decay array-type parameters into pointers. In general, use of `call_traits` is limited to advanced techniques, and will not require specializations of `call_traits` to be made. For example uses of `call_traits` see compressed_pair. For an example specialization see alloc_ptr.
24.7.5 is_empty

Answers true if the type is a class or union that has no data members, otherwise answers false. This is a key struct for determining if the space for an "empty" object can be optimized away or not.

```cpp
bool b = is_empty<T>::value;
```

24.7.6 compressed_pair

Like std::pair, but attempts to optimize away the space for either the first or second template parameter if the type is "empty". And instead of the members being accessible via the public data members first and second, they are accessible via member methods first() and second(). The `compressed_pair` handles reference types as well as other types thanks to the call_traits template. This is a good example to study if you're wanting to see how to take advantage of either call_traits or is_empty. To see an example of how compressed_pair is used see alloc_ptr.

**Remarks**

Use of the single argument constructors will fail at compile time (ambiguous call) if first_type and second_type are the same type.

The swap specialization will call swap on each member if and only if its size has not been optimized away. The call to swap on each member will look both in std, and in the member's namespace for the appropriate swap specialization. Thus clients of compressed_pair need not put swap specializations into namespace std.

A good use of compressed_pair is in the implementation of a container that must store a function object. Function objects are typically zero-sized classes, but are also allowed to be ordinary function pointers. If the function object is a zero-sized class, then the container can optimize its space away by using it as a base class. But if the function object instantiates to a function pointer, it can not be used as a base class. By putting the function object into a compressed_pair, the container implementor need not worry whether it will instantiate to a class or function pointer.
MyContainer1 uses a zero-sized Compare object. On a 32 bit machine, the sizeof MyContainer1 will be 4 bytes as the space for Compare is optimized away by compressed_pair. But MyContainer2 instantiates Compare with an ordinary function pointer which can't be optimized away. Thus the sizeof MyContainer2 is 8 bytes.

Listing: Example of compressed_pair

```cpp
#include <iostream>
#include <functional>
#include <ewl_utility>

template <class T, class Compare>
class MyContainer
{
public:
    explicit MyContainer(const Compare& c = Compare()) : data_(0, c) {}
    T* pointer() {return data_.first();}
    const T* pointer() const {return data_.first();}
    Compare& compare() {return data_.second();}
    const Compare& compare() const {return data_.second();}
    void swap(MyContainer& y) {data_.swap(y.data_);}

private:
    Metrowerks::compressed_pair<T*, Compare> data_;}

int main()
{
    typedef MyContainer<int, std::less<int>> MyContainer1;
    typedef MyContainer<int, bool (*)(int, int)> MyContainer2
    std::cout << sizeof(MyContainer1) << '
';
    std::cout << sizeof(MyContainer2) << '
';
}
```

24.7.7 alloc_ptr

An extension of std::auto_ptr. alloc_ptr will do everything that auto_ptr will do with the same syntax. Additionally alloc_ptr will deal with array new/delete:

```cpp
alloc_ptr<int, array_deleter<int>> a(new int[4]);
// Ok, destructor will use delete[]
```

Remarks

By adding the array_deleter<T> template parameter you can enable alloc_ptr to correctly handle pointers to arrays of elements.

alloc_ptr will also work with allocators which adhere to the standard interface. This comes in very handy if you are writing a container that is templated on an allocator type. You can instantiate an alloc_ptr to work with an allocator with:

```cpp
alloc_ptr<T, Allocator<T>, typename Allocator<T>::size_type> a;
```
The third parameter can be omitted if the allocator is always going to allocate and deallocate items one at a time (e.g. node based containers).

alloc_ptr takes full advantage of compressed_pair so that it is as efficient as std::auto_ptr. The sizeof(alloc_ptr<int>) is only one word. Additionally alloc_ptr will work with a reference to an allocator instead of an allocator (thanks to call_traits). This is extremely useful in the implementation of node based containers.

This is essentially the std::auto_ptr interface with a few twists to accommodate allocators and size parameters.
Chapter 25
EWL C++ Debug Mode

This chapter describes the EWL Debug Mode for code diagnostics.

25.1 Overview of EWL C++ Debug Mode

The STL portion of EWL C++ has a debug mode that can be used to diagnose common mistakes in code that uses the EWL C++ containers and their iterators.

When an error is detected, a `std::logic_error` is thrown with an appropriate error message.

25.1.1 Types of Errors Detected

Given a container (such as vector), the following errors are detected in EWL Debug mode:

- Incrementing an iterator beyond `end()`.
- Decremented an iterator before `begin()`.
- Dereferencing an iterator that it not dereferenceable.
- Any use of an invalid iterator besides assigning a valid value to it.
- Passing an iterator to a container method when that iterator does not point into that container.
- Comparison of two iterators that don't point into the same container.

25.1.2 How to Enable Debug Mode

To enable EWL C++ Debug mode simply uncomment this line in the EWL Configuration header `<ewlconfig>` See C++ Switches, Flags and Defines for more information.
Alternatively you can define _EWL_DEBUG in a prefix file. Either way, you must rebuild your C++ library after flipping this switch. Convenience makefiles are provided under ewl/EWL_C++/ to make this task easier. After rebuilding the C++ library, rebuild your application and run it. If there are any errors, a std::logic_error will be thrown. If exceptions are disabled, then instead the error function __ewl_error(const char*) is called. This function can be defined by client code. There are some sample implementations in <ewlconfig>. The default simply calls printf and abort.

25.2 Debug Mode Implementations

The debug facilities are available for the standard containers as well as the EWL extension containers.

Each container has methods that will invalidate some or all outstanding iterators. If those iterators are invalidated, then their use (except for assigning a new valid iterator) will generate an error. An iterator is considered invalidated if it no longer points into the container, or if the container's method silently causes the iterator to point to a new element within the container. Some methods (such as swap, or list::splice) will transfer ownership of outstanding iterators from one container to another, but otherwise leave them valid.

In this Example of dereference at end: the iterator i is incremented to the end of the vector and then dereferenced and assigned through. In release mode this is undefined behavior and may overwrite other important information in your application. However in debug mode this example prints out:

EWL DEBUG: dereferenced invalid iterator

Listing: Example of dereference at end:

```cpp
#include <iostream>
#include <vector>
#include <stdexcept>

int main()
{
    try
    {
        std::vector<int> v(10);
        std::vector<int>::iterator i = v.begin() + 9;
        *i = 9;  // ok
        ++i;     // ok
        *i = 10; // error
    } catch (std::exception& e)
```
{    std::cerr << e.what() << '\n'; } } 
catch (...) {    std::cerr << "Unknown exception caught\n"; } }

In the **Example of iterator/list mismatch**: an iterator is initialized to point into the first list. But then this iterator is mistakenly used to erase an element from a second list. This is normally undefined behavior. In debug mode this example prints out:

EWL DEBUG: invalid iterator given to list

**Listing: Example of iterator/list mismatch:**

```cpp
#include <iostream>
#include <list>
#include <stdexcept>

int main() {
    try {
        std::list<int> l1(10), l2(10);
        std::list<int>::iterator i = l1.begin();
        l2.erase(i);  // error
    } 
    catch (std::exception& e) {
        std::cerr << e.what() << '\n'; } 
    catch (...) {
        std::cerr << "Unknown exception caught\n"; } 
}
```

In the **Example of use of invalidated iterator**: the push_back method on deque invalidates all iterators. When the loop goes to increment `i`, it is operating on an invalidated iterator. This is normally undefined behavior. In debug mode this example prints out:

EWL DEBUG: increment end or invalid iterator

**Listing: Example of use of invalidated iterator:**

```cpp
#include <iostream>
#include <deque>
#include <stdexcept>

int main() {
    try {
        std::deque<int> d(10);
        std::deque<int>::iterator i = d.begin(), e = d.end();
        for (; i != e; ++i) 
            d.push_back(0);
    }
```
catch (std::exception& e)
{
    std::cerr << e.what() << '\n';
}
catch (...)
{
    std::cerr << "Unknown exception caught\n";
}

25.2.1 Debug Mode Containers

The list below documents when iterators are invalidated for each container, and for each method in that container:

25.2.1.1 deque

Various functions are included for debugging the deque class.

assign
All assign methods (including operator=) invalidate all iterators.

push_front/back
Invalidates all iterators.

pop_front/back
Only the iterators to the erased elements are invalidated.

insert
All iterators are invalidated.

erase
If erasing at either end, only iterators to elements erased are invalidated, else all iterators are invalidated.

resize
If the size increases, all iterators are invalidated. Else only iterators to the erased elements are invalidated.

clear
Invalidates all iterators.
swap
Iterators remain valid, but they now point into the swapped container.

Remarks
The index operator is range checked just like the at() method.

25.2.1.2 list

Various functions are included for debugging the list class.

assign
All assign methods (including operator=) invalidate all iterators.

push_front/back
No iterators are invalidated.

pop_front/back
Only the iterators to the erased elements are invalidated.

insert
No iterators are invalidated.

erase
Only the iterators to the erased elements are invalidated.

resize
Only the iterators to the erased elements are invalidated.

clear
Invalidates all iterators.

swap
Iterators remain valid, but they now point into the swapped container.

splice, merge
Iterators remain valid, but iterators into the argument list now point into this.
25.2.1.3  string

Various functions are included for debugging the string class.

assign
All assign methods (including operator=) invalidate all iterators.

push_back
If capacity is not exceeded no iterators are invalidated, else all iterators are invalidated.

pop_back
Only the iterators to the erased element is invalidated.

insert
If capacity is not exceeded iterators to elements beyond the insertion point are invalidated, else all iterators are invalidated.

erase
Iterators to elements at and beyond the erased elements are invalidated.

resize
If capacity is exceeded all iterators are invalidated, else iterators to any erased elements are invalidated.

clear
Invalidates all iterators.

swap
Iterators remain valid, but they now point into the swapped container.

Remarks
The index operator is range checked just like the at() method.

25.2.1.4  vector

Various functions are included for debugging the vector class.

assign
All assign methods (including operator=) invalidate all iterators.

**push_back**

If capacity is not exceeded no iterators are invalidated, else all iterators are invalidated.

**pop_back**

Only the iterators to the erased element is invalidated.

**insert**

If capacity is not exceeded iterators to elements beyond the insertion point are invalidated, else all iterators are invalidated.

**erase**

Iterators to elements at and beyond the erased elements are invalidated.

**resize**

If capacity is exceeded all iterators are invalidated, else iterators to any erased elements are invalidated.

**clear**

Invalidates all iterators.

**swap**

Iterators remain valid, but they now point into the swapped container.

**Remarks**

The index operator is range checked just like the at() method.

### 25.2.1.5 tree-based containers - map, multimap, set, multiset

Various functions are included for debugging the tree-based container classes map, multimap, set and multiset classes.

**assign**

Invalidates all iterators.

**insert**

No iterators are invalidated.
erase
Only the iterators to the erased elements are invalidated.

clear
Invalidates all iterators.

swap
Iterators remain valid, but they now point into the swapped container.

25.2.1.6 cdeque

Various functions are included for debugging the cdeque class.

assign
All assign methods (including operator=) invalidate all iterators.

push_front/back
If capacity exceeded invalidates all iterators, else no iterators are invalidated.

pop_front/back
Only the iterators to the erased elements are invalidated.

insert
If capacity exceeded or if insert position is not at the front or back, invalidates all iterators, else no iterators are invalidated.

erase
If erasing at either end, only iterators to elements erased are invalidated, else all iterators are invalidated.

resize
If capacity exceeded invalidates all iterators, else iterators to any erased elements are invalidated.

clear
Invalidates all iterators.

swap
Iterators remain valid, but they now point into the swapped container.

Remarks
The index operator is range checked just like the at() method.

25.2.1.7 slist

Various functions are included for debugging the slist class.

assign
All assign methods (including operator=) invalidate all iterators.

push_front/back
No iterators are invalidated.

pop_front/back
Only the iterators to the erased elements are invalidated.

insert
No iterators are invalidated.

erase
Only the iterators to the erased elements are invalidated.

resize
Only the iterators to the erased elements are invalidated.

clear
Invalidates all iterators.

swap
Iterators remain valid, but they now point into the swapped container.

splice,splice_after,merge
Iterators remain valid, but iterators into the argument list now point into this.

Remarks
Incrementing end() is not an error, it gives you begin().
25.2.1.8 hash-based containers - map, multimap, set, multiset

Various functions are included for debugging the hash based map, multimap, set and multiset classes.

assign
Invalidates all iterators.

insert
If load_factor() attempts to grow larger than load_factor_limit(), then the table is rehashed which invalidates all iterators, else no iterators are invalidated.

erase
Only the iterators to the erased elements are invalidated.

clear
Invalidates all iterators.

swap
Iterators remain valid, but they now point into the swapped container.

25.2.2 Invariants

In addition to the iterator checking described above, each container (except string) has a new member method:

bool invariants() const;

This method can be called at any time to assess the container's class invariants. If the method returns false, then the container has somehow become corrupted and there is a bug (most likely in client code, but anything is possible). If the method returns true, then no errors have been detected. This can easily be used in debug code like:

Listing: Example of invariant debugging

```c++
#include <vector>
#include <cassert>

int main()
{
    int iarray[4];
    std::vector<int> v(10);
    assert(v.invariants());
}
for (int i = 0; i <= 4; ++i)
    iarray[i] = 0;
    assert(v.invariants());

The for loop indexing over `iarray` goes one element too far and steps on the vector. The assert after the loop detects that the vector has been compromised and fires.

Be warned that the invariants method for some containers can have a significant computational expense, so this method is not advised for release code (nor are any of the debug facilities).
Chapter 26
Hash Libraries

This chapter is a reference guide to the hash support in the Embedded Warrior Libraries.

This chapter on EWL implementation of hashes is made up of the following topics. A separate chapter EWL_Utility is also useful when understanding the methodology.

- General Hash Issues
- Hash_set
- Hash_map
- Hash_fun

26.1 General Hash Issues

This document reflects issues that are common to \texttt{hash\_set, hash\_multiset, hash\_map} and \texttt{hash\_multimap}.

Rather than repeat each of these issues for each of the four hash containers, they are discussed here.

26.1.1 Introduction

These classes are analogous to \texttt{std\::set, std\::multiset, std\::map} and \texttt{std\::multimap}, but are based on a hash table. The design and implementation of these classes has the following goals:

- High CPU performance
- Minimum memory usage
- Ease of use
- Control over hashing details
• Backward compatibility with previous EWL hash containers
• Compatibility with hash containers supplied by SGI and Microsoft

Not all of these goals can be simultaneously met. For example, optimizations often require a trade-off between size and speed. "Ease of use" can pull the design in opposite directions from "control over details". And it is not possible to be 100% compatible with two or more other implementations, when they are not compatible among themselves. Nevertheless, thought and concessions have been made toward all of these goals.

26.1.2 Namespace Issues

These classes are a EWL extension to the standard C++ library, so they have been implemented within the namespace Metrowerks. There are several techniques available for accessing these classes:

26.1.2.1 Fully Qualified Reference

One technique is to fully qualify each use of an EWL extension with the full namespace. For example:

Listing: Qualified Reference

```cpp
#include <hash_set>

int main()
{
    Metrowerks::hash_set<int> a;
}
```

26.1.2.2 Namespace Alias

"Metrowerks" is a long name but it is not likely to conflict with other library's namespaces. You can easily shorten the Metrowerks namespace while still retaining the protection of namespaces through the use of an alias. For example, here is how to refer to the Metrowerks namespace as "ewl":

Listing: Namespace Alias

```cpp
#include <hash_map>

namespace ewl = Metrowerks;

int main()
{
```
ewl::hash_map<int, int> a;
}

The short name "ewl" is much more likely to conflict with other's libraries, but as the implementor of your code you can choose your aliases such that there is no conflict.

### 26.1.2.3 Using Declaration

Using declarations can bring individual names into the current namespace. They can be used either at namespace scope (outside of functions) or at function scope (inside of functions). Here is an example use of using a declaration at namespace scope:

**Listing: Namespace Scope**

```cpp
#include <hash_set>
using Metrowerks::hash_multiset;
int main()
{
    hash_multiset<int> a;
}
```

**Remarks**

Anywhere below the declaration, `hash_set` can be referred to without the use of the Metrowerks qualifier.

### 26.1.2.4 Using Directive

Using directives will import every name in one namespace into another. These can be used to essentially "turn off" namespaces so that you don't have to deal with them. They can be used at namespace scope, or to limit their effect, can also be used at function scope. For example:

**Listing: Function Scope**

```cpp
#include <hash_map>
int main()
{
    using namespace Metrowerks;
    hash_multimap<int, int> a;
}
```

**Remarks**
In the above example, any name in the Metrowerks namespace can be used in main without qualification.

26.1.2.5 Compatibility Headers

Most headers with the name <name> have an associated compatibility header <name.h>. These compatibility headers simply issue using declarations for all of the names they contain. Here is an example use:

Listing: Using Declarations for Names

```c++
#include <hash_set.h>
#include <hash_map.h>

int main()
{
    hash_set<int> a;
    hash_map<int, int> b;
}
```

26.1.2.6 Constructors

Each hash container has a constructor which takes the following arguments, with the following defaults:

```c++
size_type num_buckets = 0
const key_hasher& hash = key_hasher()
const key_compare& comp = key_compare()
float load_factor_limit = 2
float growth_factor = 4
const allocator_type& a = allocator_type()
```

Remarks

Since all arguments have defaults, the constructor serves as a default constructor. It is also declared explicit to inhibit implicit conversions from the first argument: size_type. The first argument is a way to specify the initial number of buckets. This was chosen as the first parameter in order to remain compatible both with previous versions of EWL hash containers, as well as the SGI hash containers.

The second and third parameters allow client code to initialize the hash and compare function objects if necessary. This will typically only be necessary if ordinary function pointers are being used. When function objects are used, the default constructed function object is often sufficient.
The fourth and fifth parameters allow you to set the initial values of load_factor_limit and growth_factor. Details on how these parameters interact with the size() and bucket_count() of the container can be found in the capacity section.

A second constructor also exists that accepts templated input iterators for constructing a hash container from a range. After the pair of iterators, the six parameters from the first constructor follow in the same order, and with the same defaults.

### 26.1.2.7 Iterator Issues

The hash iterators are of the forward type. You can increment them via prefix or postfix + +, but you can not decrement them. This is compatible with our previous implementation of the hash containers, and with the hash containers provided by SGI. But the hash iterators provided by Microsoft are bidirectional. Code that takes advantage of the decrement operators offered by Microsoft will fail at compile time in the EWL implementation.

**Remarks**

Forward iterators were chosen over bidirectional iterators to save on memory consumption. Bidirectional iterators would add an additional word of memory to each entry in the hash container. Furthermore a hash container is an unordered collection of elements. This "unorder" can even change as elements are added to the hash container. The ability to iterate an unordered collection in reverse order has a diminished value.

Iterators are invalidated when the number of buckets in the hash container change. This means that iteration over a container while adding elements must be done with extra care (see Capacity for more details). Despite that iterators are invalidated in this fashion, pointers and references into the hash container are never invalidated except when the referenced element is removed from the container.

### 26.1.2.8 Capacity

empty, size and max_size have semantics identical with that described for standard containers.

**Remarks**

The load factor of a hash container is the number of elements divided by the number of buckets:

```plaintext
load_factor = --------------
```

size()
bucket_count()

During the life time of a container, the load factor is at all times less than or equal to the load factor limit:

size() <= load_factor_limit()

bucket_count()

This is a class invariant. When both size() and bucket_count() are zero, the load_factor is interpreted to be zero. size() can not be greater than zero if bucket_count() is zero. Client code can directly or indirectly alter size(), bucket_count() and load_factor_limit(). But at all times, bucket_count() may be adjusted so that the class invariant is not compromised.

- If client code increases size() via methods such as insert such that the invariant is about to be violated, bucket_count() will be increased by growth_factor().
- If client code decreases size() via methods such as erase, the invariant can not be violated.
- If client code increases load_factor_limit(), the invariant can not be violated.
- If client code decreases load_factor_limit() to the point that the invariant would be violated, then bucket_count() will be increased just enough to satisfy the invariant.
- If client code increases bucket_count(), the invariant can not be violated.
- If client code decreases bucket_count() to the point that the invariant would be violated, then bucket_count() will be decreased only to the minimum amount such that the invariant will not be violated.

The final item in the bulleted list results to a "shrink to fit" statement.

myhash.bucket_count(0); // shrink to fit

The above statement will reduce the bucket count to the point that the load_factor() is just at or below the load_factor_limit().

bucket_count()

Bucket_count returns the current number of buckets in the container.

The bucket_count(size_type num_buckets) sets the number of buckets to the first prime number that is equal to or greater than num_buckets, subject to the class invariant described above. It returns the actual number of buckets that were set. This is a relatively expensive operation as all items in the container must be rehashed into the new container. This routine is analogous to vector's reserve. But it does not reserve space for a number of elements. Instead it sets the number of buckets which in turn reserves space for elements, subject to the setting of load_factor_limit().
load_factor()
returns size()/bucket_count() as a float.

load_factor_limit()
returns the current load_factor_limit.

The load_factor_limit(float lf) sets the load factor limit. If the new load factor limit is less than the current load factor limit, the number of buckets may be increased.

You can completely block the automatic change of bucket_count with:

myhash.load_factor_limit(INFINITY);

This may be important if you are wanting outstanding iterators to not be invalidated while inserting items into the container. The argument to load_factor_limit must be positive, else an exception of type std::out_of_range is thrown.

The growth_factor functions will read and set the growth_factor. When setting, the new growth factor must be greater than 1 else an exception of type std::out_of_range is thrown.

The collision(const_iterator) method will count the number of items in the same bucket with the referred to item. This may be helpful in diagnosing a poor hash distribution.

26.1.2.9  insert

Insert For Unique Hashed Containers

hash_set and hash_map

have the following insert method:

std::pair<iterator, bool>
insert(const value_type& x);

Remarks

If x does not already exist in the container, it will be inserted. The returned iterator will point to the newly inserted x, and the bool will be true. If x already exists in the container, the container is unchanged. The returned iterator will point to the element that is equal to x, and the bool will be false.

iterator insert(iterator, const value_type& x);

Operates just like the version taking only a value_type. The iterator argument is ignored. It is only present for compatibility with standard containers.
template <class InputIterator> void insert
(InputIterator first, InputIterator last);

Inserts those elements in (first, last) that don't already exist in the container.

26.1.2.10 insert

The insert for multi-hashed containers functions hash_multiset and hash_multimap have the following insert methods.

iterator insert(const value_type& x);
iterator insert(iterator p, const value_type& x);

template <class InputIterator> void insert
(InputIterator first, InputIterator last);

Remarks

In the first insert prototype x is inserted into the container and an iterator pointing to the newly inserted value is returned. If values equal to x already exist in the container, then the new element is inserted after all other equal elements. This ordering is stable throughout the lifetime of the container.

In the second prototype insert first checks to see if *p is equivalent to x according to the compare function. If it is, then x is inserted before p. If not then x is inserted as if the insert without an iterator was used. An iterator is returned which points to the newly inserted element.

The final insert prototype inserts (first, last) into the container. Equal elements will be ordered according to which was inserted first.

26.1.2.11 erase

Erases items at the position or selected items.

void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);

Remarks
The first `erase` function erases the item pointed to by position from the container. The second erases all items in the container that compare equal to `x` and returns the number of elements erased. The third `erase` erases the range (first, last) from the container.

```cpp
swap(hash_set& y);
```

Swaps the contents of *this with `y` in constant time.

```cpp
clear();
```

Erases all elements from the container.

### 26.1.2.12 Observers

Miscellaneous functions used in the hash implementation.

```cpp
get_allocator() const;
```

Returns the allocator the hash container was constructed with.

```cpp
key_comp() const
```

Returns the comparison function the hash container was constructed with.

```cpp
value_comp() const
```

Returns the comparison function used in the underlying hash table. For `hash_set` and `hash_multiset`, this is the same as `key_comp()`.

```cpp
key_hash()
```

Returns the hash function the hash container was constructed with.

```cpp
value_hash()
```

Returns the hash function used in the underlying hash table. For `hash_set` and `hash_multiset`, this is the same as `key_hash()`.

### 26.1.2.13 Set Operations

Miscellaneous hash set utility functions.
find

    iterator find(const key_type& x) const;

Returns an iterator to the first element in the container that is equal to x, or if x is not in
the container, returns end().

count

    count(const key_type& x) const

Returns the number of elements in the container equal to x.

equal_range

    std::pair<iterator, iterator> equal_range(const key_type& x);

Returns a pair of iterators indicating a range in the container such that all elements in the
range are equal to x. If no elements equal to x are in the container, an empty range is
returned.

26.1.2.14 Global Methods

Global has functions.

swap

    swap(x, y)

Same semantics as x.swap(y).

operator==

    operator == (x, y)

Returns true if x and y contain the same elements in the same order. To accomplish this
they most likely must have the same number of buckets as well.

operator!=

    operator != (x, y)

Returns !(x == y)
26.1.3 Incompatibility with Previous versions of Hash Containers

The current hash containers are very compatible with previous versions except for a few methods:

You can no longer compare two hash containers with the ordering operators: <, <=, >, >=. Since hash containers are unordered sets of items, such comparisons have little meaning.

lower_bound is no longer supported. Use find instead if you expect the item to be in the container. If not in the container, find will return end(). As there is no ordering, finding the position which an item could be inserted before has no meaning in a hash container.

upper_bound is no longer supported. Again because of the fact that hash containers are unordered, upper_bound has questionable semantics.

Despite the lack of lower_bound and upper_bound, equal_range is supported. Note that equal_range().first suffices for lower_bound, and equal_range().second suffices for upper_bound.

26.2 Hash_set

This header contains two classes: hash_set and hash_multiset.

hash_set is a container that holds an unordered set of items, and no two items in the container can compare equal. hash_multiset permits duplicate entries. Also see the General Hash Issues Introduction.

NOTE

This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

26.2.1 Introduction

These containers are in the namespace Metrowerks. See Namespace Issues for details and hints about how to best take advantage of this fact.
hash_set and hash_multiset are largely compatible with previous versions of these classes which appeared in namespace std. But see Incompatibility for a short list of incompatibilities.

26.2.2 Old Hashset Headers

Previous versions of CodeWarrior placed hash_set and hash_multiset in the headers <hashset.h> and <hashmset.h> respectively. These headers are still available, but should be used only for transition purposes. They will disappear in a future release. These headers import the contents of <hash_set> into the std namespace (as previous versions of hash_(multi)set were implemented in std.

Listing: Old Hashset Headers

```cpp
#include <hashset.h>

int main()
{
    std::hash_set<int> a;
}
```

26.2.3 Template Parameters

Both hash_set and hash_multiset have the following template parameters and defaults:

```cpp
template <class T, class Hash = hash<T>, class Compare = std::equal_to<T>, class Allocator = std::allocator<T> >
class hash_(multi)set;
```

The first parameter is the type of element the set is to contain. It can be almost any type, but must be copyable.

The second parameter is the hash function used to look up elements. It defaults to the hash function in <hash_fun>. Client code can use hash<T> as is, specialize it, or supply completely different hash function objects or hash function pointers. The hash function must accept a T, and return a size_t.

The third parameter is the comparison function which defaults to std::equal_to<T>. This function should have equality semantics. A specific requirement is that if two keys compare equal according to Compare, then they must also produce the same result when processed by Hash.

The fourth and final parameter is the allocator, which defaults to std::allocator<T>. The same comments and requirements that appear in the standard for allocators apply here as well.
26.2.4  Nested Types

hash_set and hash_multiset define a host of nested types similar to standard containers. Several noteworthy points:

- key_type and value_type are the same type and represent the type of element stored.
- key_hasher and value_hasher are the same type and represent the hash function.
- key_compare and value_compare are the same type and represent the comparison function.
- iterator and const_iterator are the same type and have semantics common to a forward const_iterator.

26.2.5  Iterator Issues

See Iterator Issues that are common to all hash containers.

Iterators of hash_set and hash_multiset are not mutable. They act as const_iterators. One can cast away the const qualification of references returned by iterators, but if the element is modified such that the hash function now has a different value, the behavior is undefined.

See Capacity for details on how to control the number of buckets.

26.2.6  hash_set

hash_set is a container based on a hash table that supports fast find, insert and erase. The elements in a hash_set are unordered. A hash_set does not allow multiple entries of equivalent elements.

26.3  Hash_map

The hash_map is a container that holds an unordered set of key-value pairs, and no two keys in the container can compare equal.

hash_multimap permits duplicate entries. Also see the General Hash Issues Introduction.

This header contains two classes:
• hash_map
• hash_multimap

NOTE
This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

26.3.1 Introduction
These containers are in the namespace Metrowerks. See Namespace Issues for details and hints about how to best take advantage of this fact.

hash_map and hash_multimap are largely compatible with previous versions of these classes which appeared in namespace std. But see Incompatibility for a short list of incompatibilities.

26.3.2 Old Hashmap Headers
Previous versions of CodeWarrior placed hash_map and hash_multimap in the headers <hashmap.h> and <hashmultimap.h> respectively. These headers are still available, but should be used only for transition purposes. They will disappear in a future release. These headers import the contents of <hash_map> into the std namespace (as previous versions of hash_(multi)map were implemented in std.

Listing: Old Hashmap Headers

```cpp
#include <hashmap.h>

int main()
{
    std::hash_map<int, int> a;
}
```

26.3.3 Template Parameters
Both hash_map and hash_multimap have the following template parameters and defaults:

Listing: Hashmap Template Parameters

```cpp
template <class Key, class T, class Hash = std::hash<Key>,
class Compare = std::equal_to<Key>,
class Allocator = std::allocator<std::pair<const Key, T>>>
```
class hash_(multi)map;

The first parameter is the type of key the map is to contain. It can be almost any type, but must be copyable.

The second parameter is the type of value that will be associated with each key. It can be almost any type, but must be copyable.

The third parameter is the hash function used to look up elements. It defaults to the hash function in <hash_fun>. Client code can use hash<Key> as is, specialize it, or supply completely different hash function objects or hash function pointers. The hash function must accept a Key, and return a size_t.

The fourth parameter is the comparison function which defaults to std::equal_to<Key>. This function should have equality semantics. A specific requirement is that if two keys compare equal according to Compare, then they must also produce the same result when processed by Hash.

The fifth and final parameter is the allocator, which defaults to std::allocator<std::pair<const Key, T>>. The same comments and requirements that appear in the standard for allocators apply here as well.

### 26.3.4 Nested Types

hash_map and hash_multimap define a host of nested types similar to standard containers. Several noteworthy points:

- key_type and value_type are not the same type. value_type is a pair<const Key, T>.
- key_hasher and value_hasher are not the same type. key_hasher is the template parameter Hash. value_hasher is a nested type which converts key_hasher into a function which accepts a value_type.
  - Value_hasher has the public typedef's

    ```
    typedef value_type argument_type;
    typedef size_type   result_type;
    ```

This qualifies it as a std::unary_function (as defined in <functional>) and so could be used where other functionals are used.

- value_hasher has these public member functions:

    ```
    size_type operator()(const value_type& x) const;
    size_type operator()(const key_type& x) const;
    ```
These simply return the result of key_hasher, but with the first operator extracting the key_type from the value_type before passing the key_type on to key_hasher.

- Key_compare and value_compare are not the same type. key_compare is the template parameter Compare. value_compare is a nested type which converts key_compare into a function which accepts a value_type.
- value_compare has the public typedef’s

```cpp
typedef value_type first_argument_type;
typedef value_type second_argument_type;
typedef bool result_type;
```

This qualifies it as a std::binary_function (as defined in <functional>) and so could be used where other functionals are used.

- value_compare has these public member functions:

```cpp
bool operator()(const value_type& x, const value_type& y) const;
bool operator()(const key_type& x, const value_type& y) const;
bool operator()(const value_type& x, const key_type& y) const;
```

These pass their arguments on to key_compare, extracting the key_type from value_type when necessary.

### 26.3.5 Iterator Issues

See Iterator Issues that are common to all hash containers.

See Capacity for details on how to control the number of buckets.

### 26.3.6 Element Access

```cpp
mapped_type& operator[](const key_type& x);
```

If the key x already exists in the container, returns a reference to the mapped_type associated with that key. If the key x does not already exist in the container, inserts a new entry: (x, mapped_type()), and returns a reference to the newly created, default constructed mapped_type.
26.4 Hash_fun

<hash_fun> declares a templated struct which serves as a function object named hash.

This is the default hash function for all hash containers. As supplied, hash works for integral types, basic_string types, and char* types (c-strings).

NOTE

This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

Client code can specialize hash to work for other types.

Alternatively, client code can simply supply customized hash functions to the hash containers via the template parameters.

The returned size_t should be as evenly distributed as possible in the range [0, numeric_limits<size_t>::max()]. Logic in the hash containers will take care of folding this output into the range of the current number of buckets.
Chapter 27
Metrowerks::threads

This chapter is a reference guide to the threads support in the Embedded Warrior Library for C++.

27.1 Overview of EWL Threads

If you're already familiar with boost::threads, then you'll be very comfortable with Metrowerks::threads. The interface closely follows the boost library. There are some minor differences.

The biggest difference is that the library is part of EWL C++, and lives in namespace Metrowerks. The entire package can be accessed via <ewl_thread>. It is essentially a fairly thin C++ wrapper over a sub-set of Posix-threads. And there is also a "single thread" version where most of the code just does nothing. It is there to ease porting multithreaded code to a single threaded environment. But be aware that your multithreaded logic may or may not translate into a working single threaded application (especially if you deal with condition variables).

The threads library currently has these configuration flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>_EWL_SINGLE_THREAD</td>
<td>A do-nothing stand-in</td>
</tr>
<tr>
<td>_EWL_USE_PTHREADS</td>
<td>Posix-Threads</td>
</tr>
<tr>
<td>_EWL_USE_WINTHREADS</td>
<td>Windows threads</td>
</tr>
</tbody>
</table>

EWL C++ will automatically configure itself based on how _EWL_THREADSAFE is set. However you can override the automatic configuration simply by setting it yourself in your prefix file or preprocessorr preference panel. You must recompile the C++ library to have the same setting.
You can now create a runtime check to make sure your EWL C++ is compiled with consistent settings:

```cpp
#include <ewl_utility>
int main()
{
    check(Metrowerks::ewl_settings());
}
```

This program will assert if it finds anything inconsistent between itself and the way EWL C++ was compiled.

### 27.2 Mutex and Locks

Metrowerks::threads has 6 types of mutexes.

- `mutex`
- `try_mutex`
- `timed_mutex`
- `recursive_mutex`
- `recursive_try_mutex`
- `recursive_timed_mutex`

#### Listing: Mutex synopsis

```cpp
class mutex
{
    public:
        typedef /* details */ scoped_lock;
        typedef /* details */ scoped_try_lock;
        void mutex();
        -mutex();
};
class try_mutex
{
    public:
        typedef /* details */ scoped_lock;
        typedef /* details */ scoped_try_lock;
        typedef /* details */ scoped_timed_lock;
        void try_mutex();
        -try_mutex();
};
class timed_mutex
{
    public:
        typedef /* details */ scoped_lock;
        typedef /* details */ scoped_try_lock;
        typedef /* details */ scoped_timed_lock;
        void timed_mutex();
        -timed_mutex();
};
class recursive_mutex
{
    public:
```
Note that each mutex type has only a default constructor and destructor. It is not copyable, and it does not have lock and unlock functions. You access this functionality via one of the nested types:

- **scoped_lock**
- **scoped_try_lock**
- **scoped_timed_lock**

**Listing: A scoped_lock**

```cpp
template <typename Mutex>
class scoped_lock
{
public:
    typedef Mutex mutex_type;
    explicit scoped_lock(mutex_type& m);
    scoped_lock(mutex_type& m, bool lock_it);
    ~scoped_lock();
    void lock();
    void unlock();
    bool locked() const;
    operator int(bool_type::*()) const;
};
```

You can use the **scoped_lock** to lock and unlock the associated mutex, and test whether it is locked or not (the operator **bool_type** is just a safe way to test the lock in an if statement like you might a pointer), for example:

```cpp```
if (my_lock) ...
```

Normally you won't use any of the **scoped_lock**'s members except it's constructor and destructor. These **lock** and **unlock** the mutex respectively.

**Listing: Example of lock and unlock usage**

```cpp```
#include <ewl_thread>
Metrowerks::mutex foo_mut;
```
void foo()
{
    Metrowerks::mutex::scoped_lock lock(foo_mut);
    // only one thread can enter here at a time
} // foo_mut is implicitly unlocked here, no matter how foo returns

In single thread mode, the above example compiles, and the lock simply doesn't do anything. If you expect foo() to call itself, or to call another function which will lock the same mutex (before foo releases foo_mut), then you should use a recursive mutex.

A mutex can conveniently be a class member, which can then be used to lock various member functions on entry. But recall that your class copy constructor will need to create a fresh mutex when copying, as the mutex itself cannot be copied (or assigned to).

In some cases you want to lock the mutex only if you don't have to wait for it. If it is unlocked, you lock it, else your thread can do something else. Use scoped_try_lock for this application. Note that not all mutex types support scoped_try_lock (have it as a nested type). The scoped_try_lock looks just like scoped_lock but adds this member function bool try_lock(),

Listing: Example of try_lock() usage

#include <ewl_thread>
Metrowerks::try_mutex foo_mut;

void foo()
{
    Metrowerks::try_mutex::scoped_try_lock lock(foo_mut, false);
    if (lock.try_lock())
    {
        // got the lock
    }
    else
    {
        // do something else
    }
}

In the above example, the second parameter in the constructor tells the lock to not lock the mutex upon construction (else you might have to wait).

Sometimes you are willing to wait for a mutex lock, but only for so long, and then you want to give up. scoped_timed_lock is the proper lock for this situation. It looks just like a scoped_lock but adds two members:

    bool timed_lock(const universal_time& unv_time);
    bool timed_lock(const elapsed_time& elps_time);

These let you specify the amount of time you're willing to wait, either in terms of an absolute time (universal_time), or in terms of an interval from the current time (elapsed_time).

Listing: Example of timed_lock()
Metrowerks::timed_mutex foo_mut;
void foo()
{
    Metrowerks::timed_mutex::scoped_timed_lock lock(foo_mut, false);
    Metrowerks::elapsed_time time_out(1, 500000000);
    if (lock.timed_lock(time_out))
    {
        // got the lock
    }
    else
    {
        // do something else
    }
}

This specifies that the thread should quit trying for the lock after 1.5 seconds. Both
elapsed_time and universal_time are simple structs with sec_ and nsec_ exposed data
members representing seconds and nanoseconds. In the case of universal_time, this is the
number of seconds and nanoseconds since midnight Jan. 1, 1970. The universal_time
default constructor returns the current time. So the above example could have also been
written as in Alternate example of timed_lock() usage.

Listing: Alternate example of timed_lock() usage

void foo()
{
    Metrowerks::timed_mutex::scoped_timed_lock lock(foo_mut, false);
    Metrowerks::elapsed_time time_out(1, 500000000);
    Metrowerks::universal_time now;
    
    if (lock.timed_lock(now + time_out))
    {
        // got the lock
    }
    else
    {
        // do something else
    }
}

In general you can add and subtract and compare universal_time and elapsed_time as makes
sense.

In single thread mode, all locks will lock their mutexes and return immediately (times are
ignored). However, if you try to lock a lockedmutex, or unlock an unlockedmutex, then an
exception of type Metrowerks::lock_error (derived from std::exception) will be thrown (even
in single thread mode).

### 27.3 Threads

The class Metrowerks::thread represents a thread of execution.

Listing: Class thread synopsis
class thread
{
public:
  thread();
  explicit thread(const std::tr1::function<void ()>& f);
  explicit thread(void (*f)());
  ~thread();

  bool operator==(const thread& rhs) const;
  bool operator!=(const thread& rhs) const;

  void join();

  static void sleep(const universal_time& unv_time);
  static void sleep(const elapsed_time& elps_time);
  static void yield();
};

A default constructed thread object represents the current thread. You can create a new thread of execution by passing a general function object, or a simple function pointer. In either case, the function must take no parameters and return void. When a thread destructs, it "detaches" the thread of execution (to use Posix-threads terminology). Once this happens, the thread is independent. You will no longer be able to refer to it, and it will clean up after itself when it terminates. But should main terminate before the thread does, the program ends anyway. You can have one thread wait on another with the join() member function.

**Listing: Example of join() function**

```cpp
#include <ewl_thread>
#include <iostream>

void do_something()
{
  std::cout << "Thread 1!";
}

int main()
{
  Metrowerks::thread t1(do_something);
  t1.join();
}
```

In the above example, main will wait for (join with) t1. Note that global objects like std::cout must be protected if more than one thread is going to access it. You must do this work yourself.

**Listing: Example of protecting threads**

```cpp
#include <ewl_thread>
#include <iostream>

Metrowerks::mutex cout_mutex;

void do_something()
{
  Metrowerks::mutex::scoped_lock lock(cout_mutex);
  std::cout << "Thread 1!";
}

void do_something_else()
```
In this example, each thread locks `cout_mutex` before using `cout`. `main()` didn't have to lock `cout` because no other threads started until after `main()` was done with `cout`.

You can also have threads sleep, but using a `mutex` and/or a condition variable (described in Condition Variables) is almost always a better solution. Similarly for `thread::yield` which is really just a convenience function for calling `sleep` with `elapsed_time(0)`.

In single thread mode, creating a thread is equivalent to a synchronous function call (though not nearly as efficient).

If you have multiple threads to create, you can create a `Metrowerks::thread_group`.

**Listing: Example of thread_group**

```cpp
class thread_group
{
public:
    thread_group();
    ~thread_group();
    thread_group();
    void create_thread(const thread::func_type& f);
    void join_all();
};
```

The main feature of `thread_group` is that it makes it very easy to join with all of the threads.

**Listing: Example of joining threads**

```cpp
int main()
{
    std::cout << "Main
    Metrowerks::thread_group g;
    g.create_thread(do_something);
    g.create_thread(do_something_else);
    g.join_all();
}
```

## 27.4 Condition Variables

A condition variable is a way for two threads to signal each other based on some predicate, such as a `queue` being empty or full. This is represented by `Metrowerks::condition`.
Condition Variables

Listing: Metrowerks::condition class synopsis

```cpp
class condition
{
    public:
        condition();
        ~condition();
        void notify_one();
        void notify_all();
        template <typename ScopedLock> void wait(ScopedLock& lock);
        template <typename ScopedLock, typename Predicate>
            void wait(ScopedLock& lock, Predicate pred);
        template <typename ScopedLock>
            bool timed_wait(ScopedLock& lock,
                const universal_time& unv_time);
        template <typename ScopedLock, typename Predicate>
            bool timed_wait(ScopedLock& lock,
                const universal_time& unv_time, Predicate pred);
        template <typename ScopedLock, typename Predicate>
            bool timed_wait(ScopedLock& lock,
                const elapsed_time& elps_time, Predicate pred);
};
```

Note that condition is not copyable nor assignable.

A condition allows one thread to pass a locked lock to the condition's wait function. The current thread then atomically unlocks the locks and goes to sleep. It will stay asleep until another thread calls this condition's `notify_one()` or `notify_all()` member function. The original thread will then atomically awake and lock the lock.

The difference between `notify_one` and `notify_all` is that the former notifies only one thread waiting on the condition, whereas the latter notifies all threads waiting on the condition.

When using the variation of the wait function without the predicate, it is important that you recheck the predicate (data) you were waiting for when the wait returns. You can not assume that whatever it is that you were wanting to be true is now true. This is most easily done by calling the wait within a while loop:

```cpp
Metrowerks::condition cond;
...
    Metrowerks::mutex::scoped_lock lock(some_mutex);
    while (I_need_more_data)
        cond.wait(lock);
```

It is up to some other thread to make `I_need_more_data` false, and it will likely need to lock `some_mutex` in order to do it. When it does, it should execute one of:

```cpp
cond.notify_one();
```

or
cond.notify_all();

It must also unlock \textit{some\_mutex} to allow the other thread's wait to return. But it does not matter whether \textit{some\_mutex} gets unlocked before or after the notification call. Once the original wakes from the wait, then the signal is satisfied. Should it wait again, then another thread will have to renotify it.

If it is more convenient, you can pass a predicate to the wait function, which will then do the while loop for you. Note that there are also several timed waits if you want to limit the sleep time (which can be thought of as an additional "condition" on the system clock).

**Example of condition usage** is a full example of condition usage. One thread puts stuff into a queue while another thread reads stuff back out of the other end.

**Listing: Example of condition usage**

```cpp
#include <iostream>
#include <queue>
#include <ewl_thread>

class unbounded_queue
{
  public:
    typedef Metrowerks::mutex Mutex;
    typedef Mutex::scoped_lock Lock;
    void send (int m);
    int receive();

  private:
    std::queue<int> the_queue_;    // The shared queue.
    Metrowerks::condition queue_is_empty_so_; // There are a lot of
    Mutex mut_;                    // conditions associated
};

void unbounded_queue::send (int m)
{
  Lock lock(mut_);
  the_queue_.push(m);
  std::cout << "sent: " << m << ' ';

  if (the_queue_.size() == 1)
    queue_is_empty_so_.notify_one();
}

int unbounded_queue::receive()
{
  Lock lock(mut_);
  while (the_queue_.empty())
    queue_is_empty_so_.wait(lock);

  int i = the_queue_.front();
  std::cout << "received: " << i << ' ';
  the_queue_.pop();
  return i;
}

unbounded_queue buf;

void sender()
{
  int n = 0;
  while (n < 1000)
  {
```

In the above example one thread continually sends data to a `std::queue`, while another thread reads data out of the queue. The reader thread must wait if the queue is empty, and the sender thread must notify the reader thread (to wake up) if the queue changes from empty to non-empty.

An interesting exercise is to transform the above example into a "bounded queue". That is, there is nothing from stopping the above example's queue from sending all of the data before the receiver thread wakes up and starts consuming it.

**Example of queue limitation** is an example if you wanted to limit the above queue to a certain number of elements (like 20).

**Listing: Example of queue limitation**

```cpp
#include <iostream>
#include <cdeque>
#include <ewl_thread>

class bounded_queue
{
public:
    typedef Metrowerks::mutex Mutex;
    typedef Mutex::scoped_lock Lock;
    typedef Metrowerks::cdeque<int> Queue;
    bounded_queue(int max) {the_queue_.reserve((unsigned)max);}
    void send (int m);
    int receive();
};
```
bool operator()() const {return c_.size() != c_.capacity();}

private:
   const C& c_; 
};

template <class C>
struct container_not_empty 
{
   container_not_empty(const C& c) : c_(c) {}
   bool operator()() const {return !c_.empty();}
private:
   const C& c_; 
};

void bounded_queue::send (int m)
{
   Lock lock(mut_);
   queue_is_full_so_.wait(lock,
      container_not_full<Queue>(the_queue_));
   the_queue_.push_back(m);
   std::cout << "sent: " << m << ' ' 
   if (the_queue_.size() == 1)
      queue_is_empty_so_.notify_one();
}

int bounded_queue::receive()
{
   Lock lock(mut_);
   queue_is_empty_so_.wait(lock,
      container_not_empty<Queue>(the_queue_));
   int i = the_queue_.front();
   std::cout << "received: " << i << ' ' 
   if (the_queue_.size() == the_queue_.capacity())
      queue_is_full_so_.notify_one();
   the_queue_.pop_front();
   return i;
}

bounded_queue buf(20);
void sender()
{
   int n = 0;
   while (n < 1000)
   {
      buf.send(n);
      ++n;
   }
   buf.send(-1);
}

void receiver()
{
   int n;
   do
   {
      n = buf.receive();
   } while (n >= 0);
}

int main()
{
   Metrowerks::thread send(sender);
   Metrowerks::thread receive(receiver);
   send.join();
}
The above example actually demonstrates more than was advertised. Not only does it limit the queue length to 20, it also introduces a non-std container (Metrowerks::cdeque) which easily enables the monitoring of maximum queue length. It also demonstrates how more than one condition can be associated with a mutex. And furthermore, it uses the predicate versions of the wait statements so that explicit while loops are not necessary for the waits. Note that the predicates are negated: the wait will loop until the predicate is true.

**Condition variables** are fairly dangerous in single threaded code. They will compile and do nothing. But note that you may loop forever waiting for a predicate that won't change:

```c++
while (the_queue.empty())
    queue_not_empty.wait(lk);
```

If `the_queue.empty()` is true then this is just an infinite loop in single thread mode. There is no other thread that is going to make the predicate false.

### 27.5 call_once

Every once in a while, you need to make sure a function is called exactly once. This is useful for initialization code for example.

The concept is similar to a local static, but local statics are not thread safe. It is possible two threads might try to construct a local static at once, before the initialization flag gets set.

**Listing: Example two threads constructing a static**

```c++
Metrowerks::mutex&
get_mutex()
{
    static Metrowerks::mutex mut; // ??!!!!
    return mut;
}
```

If more than one thread can call `get_mutex()` for the first time, at the same time, then it is possible that two threads may try to construct `mut` (and this would be bad). There are a couple of ways to deal with this problem.

You could make `mut` a global. But that may give you an undefined order of construction among global objects that is unacceptable for your application's start up code.

You could call `get_mutex()` once before you create any threads:
int main()
{
    get_mutex(); // just initialize the local static
}

Now it is safe to call get_mutex() from multiple threads as the construction step is already done.

Simple, but a little ugly. And you may not have control over main (what if you're writing a library?).

Enter Metrowerks::call_once. You can use call_once to ensure that only one thread calls get_mutex for the first time. The prototype for call_once looks like:

    void call_once(void (*func)(), once_flag& flag);

Metrowerks::once_flag is the type of flag that you must initialize (at link time) to the macro: _EWL_THREAD_ONCE_INIT.

If call_once is called with such a flag, it will atomically execute the function, and set the flag to some other value. All other threads attempting to call call_once will block until the first call returns. Later threads calling into call_once with the same flag will return without doing anything. Here is how you could use it to "initialize" get_mutex().

**Listing: Example of initializing using get_mutex()**

Metrowerks::mutex&
get_mutex_impl()
{
    static Metrowerks::mutex mut;
    return mut;
}

void init_get_mutex()
{
    get_mutex_impl();
}

Metrowerks::once_flag init_get_mutex_flag = _EWL_THREAD_ONCE_INIT;

Metrowerks::mutex&
get_mutex()
{
    Metrowerks::call_once(init_get_mutex, init_get_mutex_flag);
    return get_mutex_impl();
}

The first thread into get_mutex will also go into call_once while blocking other threads from getting past that point. It then constructs the static mutex at its leisure. Once it returns, then threads can have unfettered access to the fully constructed static mutex.

call_once works identically in single thread mode.
27.6 thread_specific_ptr

This is a way to create "thread specific data".

For example, you could create a "global" variable that is global to all functions, but local to each thread that accesses it. For example, `errno` is often implemented this way.

`Metrowerks::thread_specific_ptr` is a templated smart pointer that you can pass a new pointer to. It will associate that pointer with whatever thread passed it in (via its reset function). Other threads won't see that pointer. They will see `NULL` until they pass in their own heap-based data. The smart pointer will take care of releasing the heap data when the thread exits.

**Listing: Class thread_specific_ptr synopsis**

```cpp
template <typename T>
class thread_specific_ptr
{
    public:
        thread_specific_ptr();
        -thread_specific_ptr();
    T* get() const;
    T* operator->() const {return get();}
    T& operator*() const {return *get();}
    T* release();
    void reset(T* p = 0);
};
```

You can have as many `thread_specific_ptr`'s as you want, and pointing to whatever type you desire. The `thread_specific_ptr` is not copyable or assignable, but you can assign a pointer to it.

**Listing: Example of assigning a pointer**

```cpp
thread_specific_ptr<int> my_data;
...
my_data.reset(new int(3));
```

From then on, the thread that called reset can access that data like:

```cpp
std::cout << *my_data;
```

You can release the memory with `my_data.release()`. This transfers pointer ownership back to you, so you must then delete the pointer. But you need not call release just to prevent memory leaks. `thread_specific_ptr` will automatically delete its data. And you can put in a new pointer by calling `reset` again. `thread_specific_ptr` will make sure the original pointer gets properly deleted. Do not use the array form of `new` with `thread_specific_ptr`. It will be using `delete` to free your pointer.

**Listing: Example of freeing a pointer**

```cpp
my_data.release();
```
#include <iostream>
#include <ewl_thread>

Metrowerks::thread_specific_ptr<int> value;
void increment()
{
    ++*value;
}

Metrowerks::mutex cout_mutex;
void thread_proc()
{
    value.reset(new int(0));
    for (int i = 0; i < 1000; ++i)
        increment();

    Metrowerks::mutex::scoped_lock lock(cout_mutex);
    std::cout << *value << '
'}

int main()
{
    Metrowerks::thread_group threads;
    for (int i = 0; i < 5; ++i)
        threads.create_thread(&thread_proc);

    thread_proc();
    threads.join_all();
}

Should print out

1000
1000
1000
1000
1000
1000

Once for main, and once for the five threads. Note how no locking is necessary in accessing the "global" thread_specific_ptr. It is as if each thread has its own local copy of this global.
thread_specific_ptr
Chapter 28  
EWL std::tr1

The C++ standards committee is currently considering what will go into the next C++ standard. Several proposals have been voted into a Technical Report for consideration and the Embedded Warrior Library for C++ has some of these items in namespace `std::tr1`.

28.1 Overview of EWL Implementation of Technical Report 1

The C++ Standards Committee's report is a statement by the committee that these proposals are "interesting", but nothing more. They are not standard. They may change in the future, or even completely disappear. The EWL implementation of some of the technical report exists in namespace `std::tr1`.

You can force them to be in namespace `std` instead with:

```c
#define _EWL_TR1_NAMESPACE 0
```

EWL support consists of:

- Template class Sig class result_of
- Template class T class reference_wrapper
- Template class Sig class function
- Template class T class shared_ptr
- Template class T class enable_shared_from_this
- Template class T0, class T1, ... class T9 class tuple
- Template bind

28.2 Template class Sig class result_of
The template class `result_of` is included in the header `<functional>`.

### 28.2.1 result_of

The template class `result_of` defines a nested typedef named type, that is the return type of the function type in `result_of`'s template parameter.

**Listing: Class result_of synopsis**

```cpp
namespace std::tr1{
    template <class Sig>
    class result_of
    {
    public:
        typedef /* implementation details */ type;
    }
}
```

### 28.2.2 Public Members

The template class `result_of` provides one type defined for implementation.

#### 28.2.2.1 get_result_type

A nested typedef named type, that is the return type of the function type in `result_of`'s template parameter.

```cpp
typedef /* implementation details */ type; ()
```

**Remarks**

This can be used to specify the operator within function objects that have multiple signatures. `result_of` is typically used in template programming (as opposed to just determining the return type).

**Listing: Example usage of template class result_of**

```cpp
namespace std::tr1{
# include <functional>
# include <iostream>
# include <typeinfo>

typedef double (*FP)(int, short); 
int main()
```
{ std::cout << typeid(std::tr1::result_of<FP>::type).name() << '
';
  std::cout << typeid(result_of<less<int>(int, int)>::type).name() << '
';
};

result

double
bool

See Also

Class type_info

28.3 Template class T class reference_wrapper

The template class reference_wrapper is included in the header <functional> and is used as a wrapper around or wrapper into a container.

Listing: Class Synopsis

Namespace std::tr1 {
  template <class T> class reference_wrapper
  template <class T> reference_wrapper<T> ref(T& t)
  template <class T> reference_wrapper<const T> cref(const T& t)
}

28.3.1 reference_wrapper

The reference_wrapper is a templated wrapper class that behaves as an assignable reference.

  template <class T> class reference_wrapper

Remarks

You can pass reference_wrapper's around, and even put them into containers.

The reference_wrapper also serves as a marker for some packages that explicitly look for it, and behave differently when they find it. For example if you send a reference_wrapper to the function make_tuple, a T& will be put in the tuple instead of a T or a reference_wrapper<T>. See see the description of tuple for more details.
The template class `reference_wrapper` can also be used as a function object. It is instantiated with a function pointer or function object.

See Also
tuple, cref, ref

### 28.3.2 Public Member Functions

Public member functions that return an instance of `reference_wrapper`.

#### 28.3.2.1 ref

Returns an instance of `reference_wrapper` when passed a normal reference argument.

```cpp
template <class T> reference_wrapper<T> ref(T & t)
```

See Also
Reference_wrapper, cref()

#### 28.3.2.2 cref

Returns an instance of `reference_wrapper` when passed a const reference argument.

```cpp
template<class T> reference_wrapper<const T> cref(const T & t)
```

Listing: Example of class usage

```cpp
namespace std::tr1 {
   No example
}
```

See Also
Reference_wrapper, ref()
28.4 Template class Sig class function

The template class function is included in the header <functional>.

Remarks

This is a highly generic and powerful "call back" mechanism that includes function pointers, member function pointers, and arbitrary function objects while maintaining similar syntax and semantics to function pointers.

Listing: Class Synopsis

Namespace std::tr1 {
  Stuff
}

28.4.1 Constructors Destructors and Assignment Operator

The class provides overloaded constructors for creating and copying the class object.

28.4.1.1 Constructor

Initializes the mutex object.

```cpp
mutex();
mutex(const mutex&);
```

A default and a copy constructor are defined. The copy constructor is declared private and not defined to prevent the mutex object from being copied.

```cpp
mutex& operator=(const mutex&);
```

The assignment operator is declared private and not defined to prevent the mutex object from being copied.

28.4.1.2 Destructor

Used for implicit mutex destruction.

```cpp
~mutex();
```
Remarks

Destroys the mutex object.

Listing: Example of class usage

```cpp
#include <vector>
#include <utility>
#include <functional>
#include <iostream>
#include <cassert>

int add(int x, int y) { return x+y; }

bool adjacent(int x, int y) { return x == y-1 || x == y+1; }

struct compare_and_record
{
    std::vector<std::pair<int, int> > values;

    bool operator()(int x, int y)
    {
        values.push_back(std::make_pair(x, y));
        return x == y;
    }
};

int main()
{
    std::tr1::function <int (int, int)> f;
    f = &add;
    std::cout << f(2, 3) << 'n'; // 5
    f = std::minus<int>();
    std::cout << f(2, 3) << 'n'; // -1
    assert(f); // okay, f refers to a minus<int> object
    std::tr1::function <bool (int, int)> g;
    assert(!g); // okay, g doesn't refer to any object
    g = &adjacent;
    assert(g(2, 3)); // okay, adjacent(2, 3) returns true
    g = std::equal_to<long>(); // argument conversions ok
    assert(g(3, 3)); // okay, equal_to<long>()(3,3) returns true
    compare_and_record car;
    g = std::tr1::ref(car);
```
assert(g(3, 3)); // okay, and adds (3, 3) to car.values

g = f; // okay, int return value of f is convertible to bool

### 28.4.2 Public Member Functions

Public members that provide for mutual exclusion.

#### 28.4.2.1 Member_function

Description.

```cpp
void prototype();
```

**Listing: Example of class usage**

```
namespace std::tr1 {
#include <iostream>

    // do some stuff

int main()
{
}
}
```

### 28.5 Template class T class shared_ptr

These template shared pointer classes are included in the header `<memory>`.

### 28.6 Template class T class enable_shared_from_this

The shared_ptr is the army tank of reference counted pointers.
Its overhead is a little higher than you might be used to, but there are so many handy features (a lot of them safety features) that this pointer is hard to resist. See the proposal for a list of features (which include safety across DLL boundaries).

The shared_ptr works closely with weak_ptr, where weak_ptr takes the place of a raw pointer to break cyclic references. Again see the proposal for many more details and motivation.

This package follows closely from boost::shared_ptr, and has all of the latest improvements from that implementation.

Listing: Class Synopsis

```cpp
namespace std::tr1 {
    Stuff
}
```

### 28.6.1 Constructors Destructors and Assignment Operator

The class provides overloaded constructors for creating and copying the class object.

#### 28.6.1.1 Constructor

Initializing the mutex object.

```cpp
mutex ();
mutex(const mutex&);
```

A default and a copy constructor are defined.

The copy constructor is declared private and not defined to prevent the `mutex` object from being copied.

```cpp
mutex& operator=(const mutex&);
```

The assignment operator is declared private and not defined to prevent the `mutex` object from being copied.

#### 28.6.1.2 Destructor

Used for implicit `mutex` destruction.
Remarks

Destroys the \texttt{mutex} object.

Listing: Example of class usage

\begin{verbatim}
Namespace std::tr1 {
    #include <iostream>
   // do some stuff
    int main()
    {
    }

    result

    double
    bool
\end{verbatim}

\section*{28.6.2 Public Member Functions}

Public members that provide for mutual exclusion.

\subsection*{28.6.2.1 Member\textunderscore function}

Description.

\begin{verbatim}
void prototype();
\end{verbatim}

Listing: Example of class usage

\begin{verbatim}
Namespace std::tr1 {
    #include <iostream>   // do some stuff
    int main()
    {
    }

    result

    double
    bool
\end{verbatim}
28.7 Template class T0, class T1, ... class T9 class tuple

The header <tuple> exposes a std::pair-like class which generalizes the pair concept up to 10 members.

This proposal lives in two headers: <tuple> and <tupleio>. It closely follows these two proposals:

Remarks

Listing: Example of tuple

```cpp
#include <tuple>
#include <string>

int main()
{
    int i = 0;
    std::tr1::tuple<int, int&, std::string> t(1, i, "hi");
    // t holds (1, 0, "hi")
    i = 2;
    // t holds (1, 2, "hi")
}
```

Listing: Class Synopsis

```cpp
Namespace std::tr1 {
    Stuff
}
```

28.7.1 Constructors Destructors and Assignment Operator

The class provides overloaded constructors for creating and copying the class object.

28.7.1.1 Constructor

Initializes the mutex object.

```cpp
mutex();
mutex(const mutex&);
```

A default and a copy constructor are defined.

The copy constructor is declared private and not defined to prevent the mutex object from being copied.
mutex& operator=(const mutex&);

The assignment operator is declared private and not defined to prevent the mutex object from being copied.

### 28.7.1.2 Destructor

Used for implicit mutex destruction.

- mutex ();

Remarks

Destroys the mutex object.

**NOTE**

<tupleio> has been preserved. You must include this header to get the I/O abilities. This allows <tuple> to remain much smaller. Tuples of different sizes can be compared (==, <, etc.) with the obvious semantics. tuple_like types can be compared with each other.

**Listing: Example of <codeph> tuple </codeph> class usage**

```cpp
#include <tuple>
#include <string>
#include <utility>

int main()
{
    std::pair<int, double> p(1, 2.0);
    std::tr1::tuple<long, short, std::string> t(1, 2, "hi");
    bool b = p < t;
}
```

b gets the value true.

/* The tuples implemented here are interoperable with your own tuple_like types (should you create any).

The tuple I/O manipulators:

tuple_open
tuple_close
tuple_delimiter

take both charT arguments and const charT* arguments. Thus you can specify multi-character braces or delimiters. This can come in handy when dealing with tuples of std::string; */
```cpp
#include <tupleio>
#include <string>
#include <iostream>
#include <sstream>

int main()
{
    std::tr1::tuple<std::string, std::string> t("Hi", "5");
    std::stringstream s;
    << std::tr1::tuple_delimiter(" , ")
    << std::tr1::tuple_close(" ) ");
    s << t << 'n';
    s >> t;
    std::cout << std::tr1::tuple_open("( ")
    << std::tr1::tuple_delimiter(" , ")
    << std::tr1::tuple_close(" ) ");
    if (!s.fail())
        std::cout << t << 'n';
    else
        std::cout << "failed\n";
}

( Hi , 5 )

/* And finally, if the TR is put into namespace std (instead of std::tr1)
<tupleio> extends I/O ability to other tuple_like types such as std::pair. */

#define _EWL_TR1_NAMESPACE 0

#include <tupleio>
#include <string>
#include <iostream>
#include <map>

int main()
{
    typedef std::map<std::string, int> Map;
    Map m;
    m["one"] = 1;
    m["two"] = 2;
    m["three"] = 3;
    std::ostream_iterator<Map::value_type> out(std::cout, "\n");
    std::copy(m.begin(), m.end(), out);
}

(one 1)
(three 3)
(two 2)

28.7.2 Public Member Functions

Public members that provide for mutual exclusion.
28.7.2.1 Member_function

Description.

void prototype();

Remarks

If

Listing: Example of class usage

```cpp
namespace std::tr1 {
    #include <iostream>
    // do some stuff
    int main()
    {
    }
    
    result
    
    double
    bool
```

28.8 Template bind

The bind library is a new library voted into the first Library Technical Report.

As such, it is placed in namespace std::tr1. It is not standard, but is considered "interesting" by the C++ committee. It may become standardized in the future. This is a generalization of the existing std::bind1st and std::bind2nd functions.

The bind library consists of a series of overloaded template functions which when called will return an implementation defined function object that can then be evaluated. The purpose of bind is to adapt one function to another expected signature, or to combine simpler functions into a more complex functor. It is found in the new header <bind>. This implementation supports functions with up to 10 arguments.

Listing: Synopsis of the bind library

```cpp
namespace std { namespace tr1 {
    // 0 argument functors
    template <class F>
        // details
        bind(F f);
    
    template <class R, class F>
        // details
        bind(R r, F f);
```
bind(F f);

template <class R>
    // details
    bind(R (*)(f)());
    // 1 argument functors
    template <class F, class A1>
        // details
        bind(F f, A1 a1);
    template <class R, class F, class A1>
        // details
        bind(F f, A1 a1)
    template <class R, class B1, class A1>
        // details
        bind(R (*)(B1), A1 a1);
    template <class R, class T, class A1>
        // details
        bind(R (T::*f)(), A1 a1);
    template <class R, class T, class A1>
        // details
        bind(R (T::*f)() const, A1 a1);
    template <class R, class T, class A1>
        // details
        bind(R T::*f, A1 a1);
    // 2 argument functors
    template <class F, class A1, class A2>
        // details
        bind(F f, A1 a1, A2 a2);
    template <class R, class F, class A1, class A2>
        // details
        bind(F f, A1 a1, A2 a2);
    template <class R, class B1, class B2, class A1, class A2>
        // details
        bind(R (*)(B1, B2), A1 a1, A2 a2);
bind(R (T::*f)(B1), A1 a1, A2 a2);

template <class R, class T, class B1, class A1, class A2>
// details
bind(R (T::*f)(B1) const, A1 a1, A2 a2);
...

// 9 argument functors

bind(F f, A1 a1, A2 a2, A3 a3, A4 a4, A5 a5, A6 a6, A7 a7, A8 a8, A9 a9);

template <class R, class F, class A1, class A2, class A3, class A4,
         class A5, class A6, class A7, class A8, class A9>
// details
bind(F f, A1 a1, A2 a2, A3 a3, A4 a4, A5 a5,
     A6 a6, A7 a7, A8 a8, A9 a9);

bind(R (*f)(B1, B2, B3, B4, B5, B6, B7, B8, B9),
     A1 a1, A2 a2, A3 a3, A4 a4, A5 a5,
     A6 a6, A7 a7, A8 a8, A9 a9);

template <class R, class T, class B1, class B2, class B3, class B4,
           class B5, class B6, class B7, class B8,
           class A1, class A2, class A3, class A4, class A5,
           class A6, class A7, class A8, class A9>
// details
bind(R (T::*f)(B1, B2, B3, B4, B5, B6, B7, B8), A1 a1, A2 a2, A3 a3,
     A4 a4, A5 a5, A6 a6, A7 a7, A8 a8, A9 a9);

template <class R, class T, class B1, class B2, class B3, class B4,
           class B5, class B6, class B7, class B8,
class A1, class A2, class A3, class A4, class A5,
class A6, class A7, class A8, class A9>

// details
bind(R (T::*f)(B1, B2, B3, B4, B5, B6, B7, B8)
    const, A1 a1, A2 a2, A3 a3, A4 a4, A5 a5,
    A6 a6, A7 a7, A8 a8, A9 a9);

There are bind functions that accept a function-like object as the first argument, and then from 0 to 9 arguments. The return type of the functor can be explicitly supplied as the first template argument, or not. When not supplied, it is deduced. If the functor is a function pointer or member pointer, the return type is deduced from that signature. If the functor is a class with a nested type called result_type, the return type is deduced as F::result_type.

If the functor is a member pointer, then the first argument to the resulting functor must be a reference, pointer or smart pointer to an object containing the pointed to member. That first argument must be supplied to bind either explicitly, or implicitly as another bind expression, or a placeholder (examples will follow).

The bind overloads taking a functor with no arguments and do not accept member pointers, as member pointer functors must have at least one argument for the reference or pointer to the object.

The bind overloads taking from 1 thru 9 arguments include overloads for a function-like class, function pointers, and member function pointers.

The bind overloads taking a single argument (in addition to the functor) include an overload for taking a pointer to member data. Thus you can create a functor out of a pointer to pair<T1, T2>::first (for example).

**Listing: Simple use of std::bind2nd example**

```cpp
#include <vector>
#include <functional>
#include <algorithm>
#include <numeric>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v(10, 1);
} 
```
partial_sum(v.begin(), v.end(), v.begin());
random_shuffle(v.begin(), v.end());
ostream_iterator<int> out(cout, " ");
copy(v.begin(), v.end(), out);
cout << '
';
vector<int>::iterator i = partition(v.begin(), v.end(),
    bind2nd(less<int>(), 5));
copy(v.begin(), i, out);
cout << '
';
}

This should print out something similar to:

6 3 5 7 10 1 9 8 4 2

2 3 4 1

As stated in the introduction, bind is a generalization of bind1st and bind2nd. To transform the above example to use bind:

- Add `#include <bind>;
- Add `using namespace std::tr1;
- Add `using namespace std::tr1::placeholders;
- And finally, replace:

```
vector<int>::iterator i = partition(v.begin(), v.end(),
    bind2nd(less<int>(), 5));
```

with:

```
vector<int>::iterator i = partition(v.begin(), v.end(),
    bind(less<int>(), _1, 5));
```

Obviously bind is not a big win over bind2nd in this example. This example is meant to introduce bind by comparing it with the presumably well known std::bind2nd. Further examples will show how bind goes beyond the limitations of bind1st and bind2nd.

The `#include <bind>` is needed to bring `std::tr1::bind` into scope. As bind is a library technical report item, it lives in namespace `std::tr1` instead of in namespace `std`. 
The "_1" notation is new. The _1 is called a placeholder, it's full name is std::tr1::placeholders::_1. You can just say using namespace std::tr1::placeholders; to bring just the placeholders into scope without bringing everything else into scope. The _1 is a signal to the functor returned from bind to substitute the first argument used when calling the functor object into the place indicated by the position of the _1. That is:

```cpp
bind1st(f, x);
```

is equivalent to:

```cpp
bind(f, x, _1);
```

Both functions return a unary functor F such that F(y) calls f(x, y). In the bind example, _1 is the placeholder for y that the client will supply to F.

You can also turn f(x, y) into a function that takes no arguments by simply not using any placeholders:

```cpp
bind(f, x, y);  // -> F() calls f(x, y)
```

You can also use _1 more than once:

```cpp
bind(f, _1, _1);  // -> F(y) calls f(y, y)
```

Additionally there are more placeholders: _1, _2, _3, ... _9. You can use these placeholders to simply reorder the arguments to a function:

```cpp
bind(f, _2, _1);  // -> F(x, y) calls f(y, x)
```

And you can instruct bind to ignore parameters:

```cpp
bind(f, _2, x);  // -> F(y, z) calls f(z, x), y is ignored
```

And of course bind can handle functions (f) with a number of arguments ranging from 0 to 9. You must ensure that for every parameter of f there is a matching argument in the bind expression. Except that when f is a member function pointer, then there must be an additional argument in the bind expression, as the first argument represents the object (or a pointer to the object).
28.8.1 sort predicate

Imagine a class type Employee and the desire to sort by the member function `number()` which returns the Employee ID:

```cpp
class Employee
{
public:
    int number() const;
};

std::sort(v.begin(), v.end(),
         bind(std::less<int>(),
              bind(&Employee::number, _1),
              bind(&Employee::number, _2)
         )
);
```

The member function number is converted into a functor: once for the first argument, and once for the second argument to sort's compare predicate. Then those two bind expressions are composed as arguments to `std::less<int>`. Without bind you would have to write a custom binary compare predicate for this situation (or `operator<` for Employee).

Note that if you change Employee to:

```cpp
class Employee
{
public:
    int number;
};
```

then the predicate developed above for sorting does not change.

28.8.2 remove_if predicate

Consider a cookie factory with a quality control problem:

```cpp
class Cookie
{
public:
    Cookie(int n_chips, float diameter)
        : n_chips_(n_chips), diameter_(diameter) {}
    int number_of_chips() const {return n_chips_;
    float diameter() const {return diameter_;}

private:
    int n_chips_;  
float diameter_;  
};
```

We've got a container of cookies and we need to erase all those cookies that either have too few chips, or are too small in diameter:
v.erase(
    remove_if(v.begin(), v.end(),
        bind(logical_or<bool>(),
            bind(less<int>(), bind(&Cookie::number_of_chips, _1), 50),
            bind(less<float>(), bind(&Cookie::diameter, _1), 5.5F))
    ),
    v.end());

Note that the above continues to work whether our container holds Cookie, Cookie*, or some smart_ptr<Cookie>.

\section*{28.8.3 function}

When used in conjunction with std::tr1::function, you can store the bind expression indefinitely, and then execute it at the proper time. For example, here is a crude runtime-configurable menu example:

```cpp
#include <vector>
#include <functional>
#include <bind>
struct Document
{
    Document() {}
    Document* close() {delete this; return 0;}
    Document* save() {return this;}
    Document* print() {return this;}
};
Document* new_doc() {return new Document;}
Document* open() {return new Document;}
int main()
{
    // declare menu structure
    std::vector<std::tr1::function<Document* ()> > menu(5);
    Document* current_doc = 0;
    using std::tr1::bind;
    using std::tr1::ref;
    // load menu call backs
    menu[0] = new_doc;
    menu[1] = open;
    menu[2] = bind(&Document::close, ref(current_doc));
    menu[3] = bind(&Document::save, ref(current_doc));
    menu[4] = bind(&Document::print, ref(current_doc));
    // exercise menu call backs
    current_doc = menu[0](); // new
    current_doc = menu[2](); // close
    current_doc = menu[1](); // open
    current_doc = menu[3](); // save
    current_doc = menu[4](); // print
    current_doc = menu[2](); // close
}
```
In this example a menu is represented by a vector of functions that take no parameters and return a \document*. Installed into this menu are both namespace scope functions, and member functions bound to the current document. `std::tr1::function` is smart enough to handle both function pointers and functors (and member functions too for that matter). In this case, we bind a pointer to the object we want with the member function we want to be executed before installing it into the menu. This results in a function object that takes no parameters and returns a \document*, just like the ordinary function pointers that are also installed into the menu.

The `ref` in the bind call stands for reference. This says that instead of bind storing a copy of the pointer `current_doc` in the bind expression, store a reference to the pointer. This is done so that as the value of `current_doc` changes throughout the demo, the document upon which the menu item acts is automatically updated. In general you can wrap any argument to bind in `ref` or `cref` (`cref` is for a `const` reference) when you would like to have bind operate on the actual argument instead of a copy of it.

Alternatively `new_doc()` and `open()` could have been static functions of Document. Then their installation into the menu would have looked like:

```cpp
menu[0] = &Document::new_doc;
menu[1] = &Document::open;
```

If in the above example, an argument needed to be sent to all of the callbacks (or 2 or 3 arguments), that could easily be handled with placeholders:

```cpp
Document* print(const std::string& printer_name)
{... return this;} ...
menu[4] = bind(&Document::print, ref(current_doc), _1);...
current_doc = menu[4]("color printer");
```

So bind is really handy. And when combined with the existing algorithms in `<algorithm>` and `<numeric>`, or when combined with the new `std::tr1::function`, bind becomes ultimately flexible, and absolutely indispensable.
Chapter 29
Ewlconfig

The EWL header `<ewlconfig>` contains a description of the macros and defines that are used as switches or flags in the EWL C++ library.

29.1 C++ Switches, Flags and Defines

The EWL C++ library has various flags that may be set to customize the library to users specifications.

- _CSTD
- _Inhibit_Container_Optimization
- _Inhibit_Optimize_RB_bit
- _EWL_DEBUG
- __ewl_error
- _EWL_ARRAY_AUTO_PTR
- _EWL_CFILE_STREAM
- __EWL_CPP__
- _EWL_EXTENDED_BINDERS
- _EWL_EXTENDED_PRECISION_OUTP
- _EWL_FORCE_ENABLE_BOOL_SUPPORT
- _EWL_FORCE_ENUMS_ALWAYS_INT
- _EWL_IMP_EXP
- __EWL_LONGLONG_SUPPORT__
- _EWL_MINIMUM_NAMED_LOCALE
- _EWL_NO_BOOL
- _EWL_NO_CONSOLE_IO
- _EWL_NO_CPP_NAMESPACE
- _EWL_NO_EXCEPTIONS
- _EWL_NO_EXPLICIT_FUNC_TEMPLATE_ARG
Switches, Flags and Defines

- _EWL_NO_FILE_IO
- _EWL_NO_IO
- _EWL_NO_LOCALE
- _EWL_NO_REFCOUNT_STRING
- _EWL_NO_VECTOR_BOOL
- _EWL_NO_WCHAR
- _EWL_NO_WCHAR_LANG_SUPPORT
- _EWL_NO_WCHAR_CPP_SUPPORT
- _EWL_POSIX_STREAM
- _EWL_WIDE_FILENAME
- _EWL_WFILEIO_AVAILABLE
- _STD

29.1.1  _CSTD

The _CSTD macro evaluates to ::std if the EWL C library is compiled in the std namespace, and to nothing if the EWL C library is compiled in the global namespace.

_STD and _CSTD are meant to prefix C++ and C objects in such a way that you don't have to care whether or not the object is in std or not. For example:

_STD::cout, or _CSTD::size_t.

29.1.2  _Inhibit_Container_Optimization

If this flag is defined it will disable pointer specializations in the containers. This may make debugging easier.

You must recompile the C++ lib when flipping this switch.

29.1.3  _Inhibit_Optimize_RB_bit
Normally the red/black tree used to implement the associative containers has a space optimization that compacts the red/black flag with the parent pointer in each node (saving one word per entry). By defining this flag, the optimization is turned off, and the red/black flag will be stored as an enum in each node of the tree.

### 29.1.4 _EWL_DEBUG

This switch when enabled and the library is rebuilt will put EWL Standard C++ library into debug mode. For full information see [Overview of EWL C++ Debug Mode](#).

You must recompile the C++ lib when flipping this switch.

### 29.1.5 __ewl_error

This feature is included for those wishing to use the C++ lib with exceptions turned off. In the past, with exceptions turned off, the lib would call fprintf and abort upon an exceptional condition. Now you can configure what will happen in such a case by filling out the definition of `__ewl_error()`.

### 29.1.6 _EWL_ARRAY_AUTO_PTR

When defined auto_ptr can be used to hold pointers to memory obtained with the array form of `new`. The syntax looks like:

```c
auto_ptr<string, _Array<string> > pString(new string[3]);
pString.get()[0] = "pear";
pString.get()[1] = "peach";
pString.get()[2] = "apple";
```

Without the _Array tag, auto_ptr behaves in a standard fashion. This extension to the standard is not quite conforming, as it can be detected through the use of template arguments.

This extension can be disabled by not defining `_EWL_ARRAY_AUTO_PTR`.
29.1.7  __EWL_CFILE_STREAM

Set when the file system does not support wide character streams.

29.1.8  __EWL_CPP__

Evaluates to an integer value which represents the C++ lib's current version number. This value is best when read in hexadecimal format.

29.1.9  __EWL_EXTENDED_BINDERS

Defining this flag adds defaulted template parameters to binder1st and binder2nd. This allows client code to alter the type of the value that is stored. This is especially useful when you want the binder to store the value by const reference instead of by value to save on an expensive copy construction.

Listing: Example:

```cpp
#include <string>
#include <functional>
#include <algorithm>

struct A
{
  public:
    A(int data = 0) : data_(data) {}
    friend bool operator < (const A& x, const A& y) {return x < y;}

    private:
    int data_
    A(const A&);
};

int main()
{
  using namespace std;
  A a[5];
  A* i = find_if(a, a+5, binder2nd<less<A> >(less<A>(), A(5)));
}
```

This causes the compile-time error, because binder2nd is attempting to store a copy of A(5). But with __EWL_EXTENDED_BINDERS you can request that binder2nd store a const A& to A(5).

```cpp
A* i = find_if(a, a+5,
               binder2nd<less<A>, const A&>(less<A>(), A(5)));
```

This may be valuable when A is expensive to copy.
This also allows for the use of polymorphic operators by specifying reference types for
the operator.

This extension to the standard is detectable with template parameters so it can be disabled
by not defining _EWL_EXTENDED_BINDERS.

29.1.10 _EWL_EXTENDED_PRECISION_OUTP

When defined this allows the output of floating point output to be printed with precision
greater than DECIMAL_DIG. With this option, an exact binary to decimal conversion can be
performed (by bumping precision high enough).

The cost is about 5-6Kb in code size.

You must recompile the C++ lib when flipping this switch.

29.1.11 _EWL_FORCE_ENABLE_BOOL_SUPPORT

This tri-state flag has the following properties:

- If not defined, then the C++ library and headers will react to the settings in the
language preferences panel (as in the past).
- If the flag is set to zero, then the C++ lib/header will force "Enable bool support" to be
off while processing the header (and then reset at the end of the header).
- If the flag is set to one, then the C++ library and header will force "Enable bool
support" to be on while processing the header (and then reset at the end of the
header).

If _EWL_FORCE_ENABLE_BOOL_SUPPORT is defined, the C++ library will internally ignore the "Enable bool support" setting in the application's language preference panel, despite the fact
that most of the C++ library is compiled into the application (since it is in headers)
instead of into the binary C++ library.

The purpose of this flag is (when defined) to avoid having to recompile the C++ library
when "Enable bool" support is changed in the language preferences panel.

With _EWL_FORCE_ENABLE_BOOL_SUPPORT defined to one, std::methods will continue to have a real
bool in their signature, even when bool support is turned off in the application. But the
user won't be able to form a bool (or a true/false). The user won't be able to:

bool b = std::ios_base::sync_with_stdio(false);
// error: undefined bool and false

but this will work:

```c
unsigned char b = std::ios_base::sync_with_stdio(0);
```

And the C++ lib will link instead of getting the ctype link error.

Changing this flag will require a recompile of the C++ library.

### 29.1.12 \_EWL\_FORCE\_ENUMS\_ALWAYS\_INT

This tri-state flag has the following properties:

- If not defined, then the C++ library and headers will react to the settings in the language preference panel (as in the past).
- If the flag is set to 0, then the C++ lib/header will force "Enums always int" to be off while processing the header (and then reset at the end of the header).
- If the flag is set to 1, then the C++ library and header will force "Enums always int" to be on while processing the header (and then reset at the end of the header).

If \_EWL\_FORCE\_ENUMS\_ALWAYS\_INT is defined, the C++ library will internally ignore the "Enums always int" setting in the application's language preferences, despite the fact that most of the C++ library is compiled into the application (since it is in headers) instead of into the binary C++ library.

The purpose of this flag is (when defined) to avoid having to recompile the C++ lib when "Enums always int" is changed in the language preferences panel.

For example, with \_EWL\_FORCE\_ENUMS\_ALWAYS\_INT defined to zero, and if the user turns "enums always int" on in the language preference panel, then any enums the user creates himself will have an underlying int type.

This can be exposed by printing out the sizeof(the enum) which will be four. However, if the user prints out the sizeof(a std::enum), then the size will be one (because all std::enums fit into 8 bits) despite the enums_always_int setting in the language preference panel.

Changing this flag will require a recompile of the C++ library.

### 29.1.13 \_EWL\_IMP\_EXP

---

**EWL C++ Library Reference Manual, Rev. 10.x, 02/2014**

Freescale Semiconductor, Inc.
The C, C++, SIOUX and runtime shared libraries have all been combined into one shared library located under the appropriate OS support folder in your CodeWarrior installation path.

The exports files (.exp) have been removed. The prototypes of objects exported by the shared lib are decorated with a macro:

```
_EWL_IMP_EXP_xxx
```

where xxx is the library designation and can be defined to `__declspec(dllimport)`. This replaces the functionality of the .exp/.def files. Additionally, the C, C++, SIOUX and runtimes can be imported separately by defining the following 4 macros differently:

```
_EWL_IMP_EXP_C
_EWL_IMP_EXP_CPP
_EWL_IMP_EXP_SIOUX
_EWL_IMP_EXP_RUNTIME
```

Define these macros to nothing if you don't want to import from the associated lib, otherwise they will pick up the definition of `_EWL_IMP_EXP`.

There is a header `<UseDLLPrefix.h>` that can be used as a prefix file to ease the use of the shared lib. It is set up to import all 4 sections.

There is a problem with non-const static data members of templated classes when used in a shared lib. Unfortunately `<locale>` is full of such objects. Therefore you should also define `_EWL_NO_LOCALE` which turns off locale support when using the C++ lib as a shared lib. This is done for you in `<UseDLLPrefix.h>`. See `_EWL_NO_LOCALE` for more details.

### 29.1.14 __EWL_LONGLONG_SUPPORT__

When defined, C++ supports long long and unsigned long long integral types. Recompile the C++ lib when flipping this switch.

### 29.1.15 __EWL_MINIMUM_NAMED_LOCALE__
When defined, turns off all of the named locale stuff except for "C" and "" (which will be the same as "C"). This reduces both lib size and functionality, but only if you are already using named locales. If your code does not explicitly use named locales, this flag has no effect.

29.1.16  _EWL_NO_BOOL

If defined then bool will not be treated as a built-in type by the library. Instead it will be a typedef to unsigned char (with suitable values for true and false as well). If _EWL_FORCE_ENABLE_BOOL_SUPPORT is not defined then this flag will set itself according to the "Enable bool support" switch in the language preference panel.

The C++ lib must be recompiled when flipping this switch.

When _EWL_NO_BOOL is defined, vector<bool> will really be a vector<unsigned char>, thus it will take up more space and not have flip methods. Also there will not be any traits specializations for bool (i.e. numeric_limits).

29.1.17  _EWL_NO_CONSOLE_IO

This flag allows you to turn off console support while keeping memory mapped streams (stringstream) functional.

See Also

_EWL_NO_FILE_IO

29.1.18  _EWL_NO_CPP_NAMESPACE

If defined then the C++ lib will be defined in the global namespace.

You must recompile the C++ lib when flipping this switch.

29.1.19  _EWL_NO_EXCEPTIONS
If defined then the C++ lib will not throw an exception in an exceptional condition. Instead, `void __ewl_error(const char*)` will be called. You may edit this inline in `<ewlconfig>` to do whatever is desired. Sample implementations of `__ewl_error` are provided in `<ewlconfig>`.

**Remarks**

The operator `new` (which is in the runtime libraries) is not affected by this flag.

This flag detects the language preference panel "Enable C++ exceptions" and defines itself if this option is not on.

The C++ lib must be recompiled when changing this flag (also, if the language preference panel is changed).

### 29.1.20  _EWL_NO_EXPLICIT_FUNC TEMPLATE_ARG

When defined, assumes that the compiler does not support calling function templates with explicit template arguments.

On Windows, when "Legacy for-scoping" is selected in the language preference panel, then this switch is automatically turned on. The Windows compiler goes into a MS compatible mode with ARM on.

This mode does not support explicit function template arguments. In this mode, the signatures of `has_facet` and `use_facet` change.

You must recompile the C++ lib when flipping this switch.

**Listing: Example of _EWL_NO_EXPLICIT_FUNC_TEMPLATE_ARG usage:**

```cpp
Standard setting:
template <class Facet>
    const Facet& use_facet(const locale& loc);

template <class Facet>
    bool has_facet(const locale& loc) throw();

_EWL_NO_EXPLICIT_FUNC_TEMPLATE_ARG setting.
template <class Facet>
    const Facet& use_facet(const locale& loc, Facet*);

template <class Facet>
    bool has_facet(const locale& loc, Facet*) throw();
```
29.1.21  _EWL_NO_FILE_IO

This flag allows you to turn off file support while keeping memory mapped streams (stringstream) functional.

See Also

_EWL_NO_CONSOLE_IO

29.1.22  _EWL_NO_IO

If this flag is defined, C++ will not support any I/O (not even stringstreams).

29.1.23  _EWL_NO_LOCALE

When this flag is defined, locale support is stripped from the library. This has tremendous code size benefits.

All C++ I/O will implicitly use the "C" locale. You may not create locales or facets, and you may not call the imbue method on a stream. Otherwise, all streams are completely functional.

The C++ lib must be recompiled when flipping this switch.

29.1.24  _EWL_NO_REFCOUNT_STRING

The flag _EWL_NO_REFCOUNT_STRING is deprecated and will have no effect (it is harmless). This rewrite has higher performance and lower code size compared to previous releases.

29.1.25  _EWL_NO_VECTOR_BOOL
If this flag is defined it will disable the standard `vector<bool>` partial specialization. You can still instantiate `vector<bool>`, but it will not have the space optimization of one `bool` per bit.

There is no need to recompile the C++ lib when flipping this switch, but you should remake any precompiled headers you might be using.

### 29.1.26 _EWL_NO_WCHART

This flag has been replaced by three new flags:

- `_EWL_NO_WCHART_LANG_SUPPORT`
- `_EWL_NO_WCHART_C_SUPPORT`
- `_EWL_NO_WCHART_CPP_SUPPORT`

### 29.1.27 _EWL_NO_WCHART_LANG_SUPPORT

This flag is set if the compiler does not recognize `wchar_t` as a separate data type (no `wchar_t` support in the language preference panel). The C++ lib will still continue to support wide character functions. `wchar_t` will be typedef’d to another built-in type.

The C++ library must be recompiled when turning this switch on (but not when turning it off).

### 29.1.28 _EWL_NO_WCHART_C_SUPPORT

This flag is set if the underlying C lib does not support wide character functions. This should not be set when using EWL C.

The C++ library must be recompiled when turning this switch on (but not when turning it off).

### 29.1.29 _EWL_NO_WCHART_CPP_SUPPORT
This flag can be set if wide character support is not desired in the C++ lib. Setting this flag can cut the size of the I/O part of the C++ lib in half.

The C++ library must be recompiled when turning this switch on (but not when turning it off).

### 29.1.30 _EWL_POSIX_STREAM

Set when a POSIX based library is being used as the underlying C runtime library.

### 29.1.31 _EWL_WIDE_FILENAME

If the flag _EWL_WIDE_FILENAME is defined, then the file stream classes support wide character filenames (null terminated arrays of \texttt{const wchart_t*}). Each stream class has an overloaded constructor, and an overloaded open member taking the \texttt{const wchar_t}. If the underlying system supports wide filenames, EWL C++ will pass the \texttt{wchar_t} straight through without any locale encoding.

Thus the interpretation of the wide filename is done by the OS, not by the C++ library. If the underlying system does not support wide filenames, the open will fail at runtime.

By default _EWL_WIDE_FILENAME is not defined as these signatures are not standard.

Turning on this flag does not require a recompile of EWL C++.

When EWL C is not being used as the underlying C library, and when the file stream is implemented in terms of \texttt{FILE*} (see _EWL_CFILE_STREAM ), the system is said to not support wide filenames and the open will fail at runtime.

When using Posix as the underlying implementation (see _EWL_POSIX_STREAM ), wide filenames are supported if the Posix library comes from the EWL Extras Library (in which case the _EWL_WFILEIO_AVAILABLE flag must be on).

### 29.1.32 _EWL_WFILEIO_AVAILABLE

Set when a wide character file name is available for a file name.
29.1.33  _STD

This macro evaluates to ::std if the C++ lib is compiled in the std namespace, and to nothing if the C++ lib is compiled in the global namespace.

SeeAlso

_CSTD
Index

_AEWL_CPP_  668
__ewl_error  667
__EWL_LONGLONG_SUPPORT__  671
_CSTD  666
_EWL_ARRAY_AUTO_PTR  667
_EWL_CFILE_STREAM  668
_EWL_CXLIMITED_RANGE  375
_EWL_DEBUG  667
_EWL_EXTENDED_BINDERS  668
_EWL_EXTENDED_PRECISION_OUTPUT  669
_EWL_FORCE_ENABLE_BOOL_SUPPORT  669
_EWL_FORCE_ENUMS_ALWAYS_INT  670
_EWL_IMP_EXP  670
_EWL_MINIMUM_NAMED_LOCALE  671
_EWL_NO_BOOL  672
_EWL_NO_CONSOLE_IO  672
_EWL_NO_CPP_NAMESPACE  672
_EWL_NO_EXCEPTIONS  672
_EWL_NO_EXPLICIT_FUNC_TEMPLATE_ARG  673
_EWL_NO_FILE_IO  674
_EWL_NO_IO  674
_EWL_NO_LOCALE  674
_EWL_NO_REFCOUNT_STRING  674
_EWL_NO_VECTOR_BOOL  674
_EWL_NO_WCHART  675
_EWL_NO_WCHART_C_SUPPORT  675
_EWL_NO_WCHART_CPP_SUPPORT  675
_EWL_NO_WCHART_LANG_SUPPORT  675
_EWL_POSIX_STREAM  676
_EWL_RAW_ITERATORS  321
_EWL_WFILEIO_AVAILABLE  676
_EWL_WIDEFILENAME  676
_InhibitContainerOptimization  666
_InhibitOptimizeRB_bit  666
_STD  677
<cmath>  373
<cstdlib>  373
<sstream>  511

A

Abnormal Termination  93
abort  82
abbrev_monthname  229
abbrev_weekday  229
abs  383
Access  576
accumulate  370
Adaptors  116, 117
Addition Operator  143
address  121
adjacent_difference  372
adjacent_find  325
advance  306
Algorithms  125
alloc_ptr  594
allocate  121
allocator_type  569
allocator members  121
Allocator Requirements  104
always_noconv  192
am_pm  230
any  299
append  148
apply  538
Arbitrary-Positional Stream  58
arg  384
Arithmetic operations  109
array_size  591
Array Forms  84
Assertions  100
assign  136, 148, 275, 277, 285
Assignment  573
Assignment Operator  143, 353
Assignments  141
Associative Containers  287
at  147
atexit  82
auto_ptr  126, 129–132
auto_ptr_ref  132

B

back_insert_iterator  311
Back_insert_iterator Operators  311
back_inserter  312
bad  429
bad_alloc  86
Bad_alloc  86
bad_cast  89
bad_exception  92
bad_typeid  90
base  307
basic_filebuf::close  531
basic_filebuf::imbue  534
basic_filebuf::is_open  529
basic_filebuf::open  530
basic_filebuf::overflow  532
basic_filebuf::pbackfail  532
basic_filebuf::seekoff  533
basic_filebuf::seekpos  533
basic_filebuf::setbuf  533
basic_filebuf::showmanyc  531
basic_filebuf::sync  534
basic_filebuf::underflow  532
Basic Iterator 305
before 89
begin 144
Bidirectional Iterators 304
binary_negate 114
binary_search 340
bind1st 115
bind2nd 116
binder1st 115
binder2nd 115
Binders 115
bind Library 655
bitset Constructor 294
Bitset Members 294
Bitset Operators 300
bitvector Destructor 573
bool Operator 422
bsearch 348
Buffer Management Positioning 454
Byte strings 63

C
c_str 151
C++ 665
C++ Library 57
call_once 638
call_traits 592
can_derive_from 592
capacity 146, 285, 575
Capacity 145, 574
Case Transformation 185
cdequeue 604
char_traits 139
class 136
Character 58
Character Classification 177
Character Conversions 178
Character sequences 63
Character Sequences 58
Character Support 161
Character Trait Definitions 136
Character traits 135
Class 463, 489
classic 176
classic_table 190
Classification 184
clear 146, 279, 425, 579
C Library Locales 270
close 263
Codecv 191
codecv_byname 194
Codecv_byname 195
codecv Members 191
codecv Virtual Functions 193
collate_byname 208
collate Data Section 208
collate Virtual Functions 207
combine 174
compare 137, 155, 206
Comparison Function 58
Comparisons 110
compile_assert 591
complex Class 375
complex Class Operators 378
complex Member Functions 377
complex Template Class 376
Component 59
compressed_pair 593
Condition Variables 633
conj 384
const_mem_fun_ref 1120
const_mem_fun_t 119
const_mem_fun1_ref_t 120
const_mem_fun1_t 119
const_reference 311, 571
Constraints 67
construct 122
Constructor 86, 189
Constructors 88--92, 107, 141, 173
Container Adaptors 281
containers 603
Containers 271
Contraction 211
Conversion Constructor 131
copy 138, 150, 328
copy_backward 328
Copy Construction 104
copyfmt 422
cos 385
cosh 385
count 297, 326
count_if 326
cref 646
cshift 358
c_type 179
c_type_base 179
c_type_byname 183
c_type_byname<Char> 191
c_type<Char> Static Members 190
c_type Members 189
c_type Specializations 188
c_type Virtual Functions 182
curr_symbol 252, 259

data 151
Data file syntax 258
date 230
Date 132
date_time 230
deallocate 121
decimal_point 201, 204, 251, 258
default_century 231
Default Construction 104
Defines 665
Definitions 57
delete 84
delete[] 85
denorm_min 78
deque 600
deque Constructor 274
deque Template Class 274
Derived classes 68
destroy 122
destructor 86, 91, 130, 173, 189, 316
Destructor 143
Diagnostics Library 97
digits 73
Directive 611
distance 306
divides 110
do_is 182
do_narrow 183
do_scan_is 182
do_scan_not 182
do_tolower 183
do_toupper 182
do_widen 183
domain_error 98
Dynamic Memory Management 83

E
Element Access 147
empty 146, 283, 574
Enable Debug Mode 597
end 144
eof 139, 427
epsilon 74
eq 137
eq_int_type 139
equal 319, 327
equality 289, 291, 340
equal_to 111
equality Comparisons 103
erase 149, 276, 278, 286, 579, 616
Erasure 579
Error Numbers 101
Escape sequences 167
EUC 196
EWL 643
EWL_Utility 583
EWL C++ Library 57
Ewlconfig 665
EWL Threads 627
exception 91
Exception Classes 97
Exception Handling 91
exceptions 431
exit 82
exp 386
Expansion 211
Extending derivation 235
External Linkage 68
extractors 160

F
fail 428
failed 320
failure 403
failure::what 403
false_name 205
falsename 202
fill 331, 421
fill_n 332
find 137, 152, 288, 290, 324
find_end 324
find_first_not_of 153
find_first_of 152, 325
find_if 324
find_last_not_of 154
find_last_of 153
flags 407
Flags 665
flip 297, 581
float_denorm_style 80
float_round_style 80
fmtflags 403
for_each 323
Format Parsing 220
Forward Iterators 304
fpos Template Class 401
frac_digits 253, 260
Freestanding Implementations 66
freeze 553, 562, 565
French collation 211
front 576
front_insert_iterator 312
Front_insert_iterator operators 313
front_insert_iterator Template Class 312
front_inserter 313
fstream Header 527
Function objects 108

G
General Utilities Libraries 103
generate 332
generate_n 332
get 131, 198, 247, 263
get_allocator 151, 575
get_result_type 644
get_state 139
get_temporary_buffer 125
getline 161
getloc 413
global 176
good 427
greater 111
greater_equal 112
grouping 201, 204, 252, 259
gslice 365
gslice_array 366
gslice_array Template Class 366
gslice Access Functions 365

H
Handler function 69
Handler Function 59
has_denorm 76
has_denorm_loss 77
has_facet 177
has_infinity 76
hasQuiet_NaN 76
has_signaling_NaN 76
hash 207
hash_fun 625
hash_map 621
hash_set 619, 621
Hash Libraries 609
Header 401
Header Algorithm 323
Header_iterator 305
Headers 65, 67, 68

I
Ignorable Characters 212
imag 378, 383
imbue 413, 420
Implementation messages 264
in 180, 192
includes 341
indirect_array 369
indirect_array Template Class 369
infinity 77
Init Class Constructor 406
inner_product 371
inplace_merge 341
Input/Output Manipulations 162
Input Iterators 303
insert 149, 275, 277, 286, 578, 615, 616
insert_iterator 314
insert_iterator Operators 314
inserter 315
Inserters 160
Insertion 577
Insert Iterators 311
invalid_argument 98
invariants 581
Invariants 606

ios 401
ios_base 416
ios_base Class 402
ios_base Constructor 416
ios_base Destructor 416
ios_base Format Flags 407
ios_base Locale Functions
ios_base Storage Function 414
iosfwd 391
iostream 404
iostream Access Functions 395
Iostream Class Templates 59
Iostream Objects 395
Iostreams Base Classes 401
is_bounded 78
is_const 587
is_empty 593
is_exact 74
is_iec559 78
is_integer 73
is_modulo 78
is_same 587
is_signed 73, 589
is_specialized 72
is_unsigned 589
is_volatile 588
istream_iterator 316
istream_iterator Operations 316
istream_iterator Template Class 315
istreambuf_iterator 318
istreambuf_iterator Operators 318
istreambuf_iterator Template Class 318
iostream cin 396
istream Class 558
istream Destructor 559
iter_swap 329
Iteration 576
Iterators Library 303
Iterator Support 144
Iterator Traits 305
iword 414

J
JIS 196

L
Language Support Library 71
length 137, 145, 192
length_error 99
less 112
less_equal 112
lexicographical_compare 347
library 67
Library-wide Requirements 65
Linkage 67
list 601
   list Template Class 276
locale 169
   locale::Category 171
locale::facet 172
locale::id 172
Locale Globals 176
Locale Members 174
Locale Names 170
Locale Operators 174
Locales 169, 439, 454
Locale Static Members 175
Locale Types 171
Localization Library 165
Locks 628
log 386
log10 386
logic_error 98
logical_and 113
logical_not 113
logical_or 113
Logical operations 113
lower_bound 288, 291, 339
lt 137

M
make_heap 344
make_pair 108
map, 603
Map Operations 288
map Template Class 287
mask_array 367, 368
mask_array Template Class 367
max 73, 345, 357
max_element 346
max_exp 75
max_exp10 75
max_length 192
max_size 122, 145, 575
mem_fun 118
mem_fun_ref 119
mem_fun_ref_t 118
mem_fun_t 117
mem_fun1_ref_t 118
mem_fun1_t 117
Memory 120
merge 280, 341
messages_byname 266
Messages Virtual Functions 263
min 72, 345, 357
min_element 346
min_exp 75
min_exp10 75
minus 109
mismatch 326
Modifier 59, 209

Modifiers 147
modulus 110
money_get 246
Money_get Members 247
Money_get Virtual Functions 248
money_put 248
Money_put Members 249
Money_put Virtual Functions 249
Money class 241
Moneypunct 250
moneypunct_byname 256
moneypunct Derivation 255
Moneypunct Members 251
Moneypunct Virtual Functions 254
monthname 229
move 138
Multibyte strings 64
multimap, 603
multimap Constructor 290
multimap Template Class 290
multiplies 109
multiset 603
multiset Constructor 293
multiset Template Class 293
Multi-Thread Safety 61
Mutex 628

N
name 89, 174
narrow 181
Narrow-oriented Iostream Classes 60
Narrow stream objects 396
neg_format 254, 260
negate 110
negative_sign 252, 259
Negators 114
new 84
new_handler 87
new[] 84
next_permutation 347
noconv 195
none 299
Non-Member Functions 155
Non-member Logical Operations 361
norm 384
not_eof 138
not_equal_to 111
NTCTS 60, 136
nth_element 339
Null Terminated Sequence Utilities 161
Num_get Members 198
Num_get Virtual Functions 199
Num_put Members 200
Num_put Virtual Functions 200
Numeric wide 205
Numeric arrays 352
Numeric limits 72
Numerics Library 351
numpunctbyname 203
numpunct Derivation 205
Numpunct Members 201
Numpunct virtual functions 202

O
open 263
openmode 405
operator 107, 124, 129, 174, 175, 281
operator!= 88, 106, 123, 157, 298
operator^ 300
operator^= 295
operator~ 296
operator[] 147, 288, 354
operator* 130
Operator* 312
operator& 300
operator&= 294
operator+ 155
operator++ 124
Operator++ 312
operator+= 147
operator< 157, 282
operator<< 160, 299, 301
Operator<< 490
operator<<= 295
operator<= 106, 158
operator= 124
operator== 88, 122, 156, 298
operator> 106, 158
operator-> 130
operator>> 107, 159
operator>> 160, 299, 301
Operator>> 464
operator>>= 295
operator| 300
operator| 295
Operators 106
ostream_iterator 317
ostream_iterator Operators 317
ostream_iterator Template Class 317
ostreambuf_iterator 320
ostreambuf_iterator Operators 320
ostreambuf_iterator Template Class 319
ostream cerr 397
ostream clog 397
ostream cout 397
ostream Class 560
ostream Destructor 561
out 191
out_of_range 99
Output Iterators 304
overflow 557
overflow_error 100

Overloaded complex Operators 380

P
Pairs 107
partial_sort 338
partial_sort_copy 338
partial_sum 371
partition 336
pbackfail 557
pcount 554, 562, 565
Placement Forms 85
Placement operator delete 85
Placement operator new 85
plus 109
pointer_to_binary_function 116
pointer_to_unary_function 116
Pointers 116, 117
polar 384
pop 283, 284
pop_back 279, 579
pop_front 278
pop_heap 344
pos_format 253, 260
positive_sign 252, 259
pow 387
precision 411
Predefined Iterators 307
prev_permutation 347
priority_queue 282
Private members 64
Protected Member Functions 450
push 282, 284
push_back 278, 577
push_front 278
push_heap 344
put 200, 249
Putback 447
pword 415

Q
qsort 348
Query 587
quiet_NaN 77

R
radix 74
random_shuffle 336
Random Access Iterators 304
range_error 99
raw_storage_iterator 123
Raw storage iterator 123
rbegin 144
rbuf 419, 559, 563, 565
rdstate 423
<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>378, 383</td>
</tr>
<tr>
<td>ref</td>
<td>646</td>
</tr>
<tr>
<td>reference</td>
<td>570</td>
</tr>
<tr>
<td>reference_wrapper</td>
<td>645</td>
</tr>
<tr>
<td>register_callback</td>
<td>415</td>
</tr>
<tr>
<td>Relation</td>
<td>209</td>
</tr>
<tr>
<td>Relationals</td>
<td>210</td>
</tr>
<tr>
<td>release</td>
<td>131</td>
</tr>
<tr>
<td>remove 279, 332</td>
<td></td>
</tr>
<tr>
<td>remove_all 586</td>
<td></td>
</tr>
<tr>
<td>remove_bounds 586</td>
<td></td>
</tr>
<tr>
<td>remove_const 584</td>
<td></td>
</tr>
<tr>
<td>remove_copy 333</td>
<td></td>
</tr>
<tr>
<td>remove_copy_if 333</td>
<td></td>
</tr>
<tr>
<td>remove_cv 585</td>
<td></td>
</tr>
<tr>
<td>remove_if 280, 333</td>
<td></td>
</tr>
<tr>
<td>remove_pointer 585</td>
<td></td>
</tr>
<tr>
<td>remove_reference 586</td>
<td></td>
</tr>
<tr>
<td>remove_volatile 584</td>
<td></td>
</tr>
<tr>
<td>rend 145</td>
<td></td>
</tr>
<tr>
<td>replace 150, 330</td>
<td></td>
</tr>
<tr>
<td>replace_copy 330</td>
<td></td>
</tr>
<tr>
<td>replace_copy_if 331</td>
<td></td>
</tr>
<tr>
<td>Replacement Function</td>
<td>60, 68</td>
</tr>
<tr>
<td>Repositional Stream</td>
<td>60</td>
</tr>
<tr>
<td>reserve 146, 575</td>
<td></td>
</tr>
<tr>
<td>Reserved Function</td>
<td>61</td>
</tr>
<tr>
<td>Reserved Names</td>
<td>67</td>
</tr>
<tr>
<td>reset 131, 296</td>
<td></td>
</tr>
<tr>
<td>Reset 209</td>
<td></td>
</tr>
<tr>
<td>resetiosflags 505</td>
<td></td>
</tr>
<tr>
<td>resize 146, 275, 277, 286, 358, 580</td>
<td></td>
</tr>
<tr>
<td>result_of 643, 644</td>
<td></td>
</tr>
<tr>
<td>return_temporary_buffer</td>
<td>125</td>
</tr>
<tr>
<td>reverse 280, 335</td>
<td></td>
</tr>
<tr>
<td>reverse_copy 335</td>
<td></td>
</tr>
<tr>
<td>reverse_iterator 307</td>
<td></td>
</tr>
<tr>
<td>reverse_iterator Operators</td>
<td>308</td>
</tr>
<tr>
<td>Reverse Iterators</td>
<td>307</td>
</tr>
<tr>
<td>rfind 152</td>
<td></td>
</tr>
<tr>
<td>rotate 335</td>
<td></td>
</tr>
<tr>
<td>rotate_copy 335</td>
<td></td>
</tr>
<tr>
<td>round_error 74</td>
<td></td>
</tr>
<tr>
<td>round_style 79</td>
<td></td>
</tr>
<tr>
<td>slist</td>
<td>605</td>
</tr>
<tr>
<td>sort 280, 337</td>
<td></td>
</tr>
<tr>
<td>sort_heap 345</td>
<td></td>
</tr>
<tr>
<td>Sorting Operations</td>
<td>337</td>
</tr>
<tr>
<td>sort predicate 661</td>
<td></td>
</tr>
<tr>
<td>splice 279</td>
<td></td>
</tr>
<tr>
<td>sqrt 388</td>
<td></td>
</tr>
<tr>
<td>stable_partition 337</td>
<td></td>
</tr>
<tr>
<td>stable_sort 337</td>
<td></td>
</tr>
<tr>
<td>stack Template Class</td>
<td>283</td>
</tr>
<tr>
<td>Standard Locale Categories</td>
<td>178</td>
</tr>
<tr>
<td>start 81</td>
<td></td>
</tr>
<tr>
<td>std::tr1 643</td>
<td></td>
</tr>
<tr>
<td>Storage Allocation</td>
<td>83</td>
</tr>
<tr>
<td>Storage Allocation Errors</td>
<td>86</td>
</tr>
<tr>
<td>Storage Deallocation</td>
<td>83</td>
</tr>
<tr>
<td>str 554, 560, 563, 566</td>
<td></td>
</tr>
<tr>
<td>Stream Buffering</td>
<td>395</td>
</tr>
<tr>
<td>Stream Buffers</td>
<td>437</td>
</tr>
<tr>
<td>Stream Iterators</td>
<td>315</td>
</tr>
<tr>
<td>stride 363, 366</td>
<td></td>
</tr>
<tr>
<td>string 602</td>
<td></td>
</tr>
<tr>
<td>String-based Streams</td>
<td>511</td>
</tr>
<tr>
<td>String Classes</td>
<td>140</td>
</tr>
</tbody>
</table>
stringfwd 391
String Operations 151
Strings Library 135
String Support 162
String Syntax 169
Strstream 551
strstreambuf Class 551
strstreambuf Destructor 553
strstream Class 564
strstream Destructor 564
struct char_traits<T> 140
substr 154
sum 357
Supported Locale Names 165
swap 151, 160, 276, 281, 291–293, 329, 581
swap_ranges 329
Switches, 665
sync_with_stdio 415
SZ_T 390
T
tan 388
tanh 388
Template Class 126
Temporary buffers 125
terminate 94
terminate_handler 93
termination 81
test 298
Text-Argument: 209
thousands_sep 201, 204, 251, 258
thread_specific_ptr 640
Threads 631
Thread Safety Policy 62
tie 418
time 230
Time 132
time_12hour 230
time_get 217, 218
time_getbyname 225
time_put 226
time_putbyname 227
time_put Virtual Functions 226
time_zone 231
timepunctbyname 237
tininess_before 79
to_char_type 138
to_int_type 139
to_string 297
to_ulong 297
tolower 178, 181
top 284
toupper 178, 181
traits 136
Traits 61
transform 207, 330
traps 79
tree-based 603
true_name 205
true_name 201
type_info 88
typedef Declarations 402
Type identification 87
Types 71
U
UCS-2 195
Unary negate 114
uncaught_exception 94
underflow 556
underflow_error 100
unexpected 93
unexpected_handler 93
uninitialized_copy 125
uninitialized_fill 126
uninitialized_filln 126
unique 280, 334
unique_copy 334
unsetf 410
unshift 192
upper_bound 289, 339
use_facet 176
utc_offset 231
UTF-8 196
Utility Components 106
V
valarray 353
valarray Binary Operators 359
valarray Destructor 353
valarray Logical Operators 360
valarray Member Functions 356
valarray Template Class 352
valarray Unary Operators 354
value_type 570
vector 602
Vector<bool> 287
vector Template Class 284
W
weekday 229
what 86, 90, 92
Wide-character sequences 64
widen 181
Wide-oriented IOSTREAM Classes 61
Wide stream objects 398
width 412
wistream wcin 398
wostream wcerr 398
wostream wcout 398
wostream wlcog 399

x

xalloc 414
Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo and CodeWarrior are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2009–2016 Freescale Semiconductor, Inc. All rights reserved.