5.5. Auxiliary Functions

The C++ standard library provides some small auxiliary functions that process minimum and maximum, swap values, or provide supplementary comparison operators.

5.5.1. Processing the Minimum and Maximum

Table 5.18 lists the utility functions `<algorithm>` provides to process the minimum and/or maximum of two or more values. All `minmax()` functions and all function for initializer lists are provided since C++11.

Table 5.18. Operations to Process Minimum and Maximum

Operation	Effect
`min(`*a*`,`*b*`)`	Returns the minimum of *a* and *b*, comparing with <
`min(`*a*`,`*b*`,`*cmp*`)`	Returns the minimum of *a* and *b*, comparing with *cmp*
`min(`*initlist*`)`	Returns the minimum in *initlist*, comparing with <
`min(`*initlist*`,`*cmp*`)`	Returns the minimum in *initlist*, comparing with *cmp*
`max(`*a*`,`*b*`)`	Returns the maximum of *a* and *b*, comparing with <
`max(`*a*`,`*b*`,`*cmp*`)`	Returns the maximum of *a* and *b*, comparing with *cmp*
`max(`*initlist*`)`	Returns the maximum in *initlist*, comparing with <
`max(`*initlist*`,`*cmp*`)`	Returns the maximum in *initlist*, comparing with *cmp*
`minmax(`*a*`,`*b*`)`	Returns the minimum and maximum of *a* and *b*, comparing with <
`minmax(`*a*`,`*b*`,`*cmp*`)`	Returns the minimum and maximum of *a* and *b*, comparing with *cmp*
`minmax(`*initlist*`)`	Returns the minimum and maximum of *initlist* comparing with <
`minmax(`*initlist*`,`*cmp*`)`	Returns the minimum and maximum of *initlist* comparing with *cmp*

The function `minmax()` returns a `pair<>` (see Section 5.1.1, page 60), where the first value is the minimum, and the second value is the maximum.

For the versions with two arguments, `min()` and `max()` return the first element if both values are equal. For initializer lists, `min()` and `max()` return the first of multiple minimum or maximum elements. `minmax()` returns the pair of `a` and `b` for two equal arguments and the first minimum but the last maximum element for an initializer list. However, it is probably a good programming style not to rely on this.

Note that the versions taking two values return a reference; the versions taking initializer lists return copies of the values:

```
namespace std {
    template <typename T>
      const T& min (const T& a, const T& b);
    template <typename T>
      T min (initializer_list<T> initlist);
    ...
}
```

The reason is that for an initializer list, you need an internal temporary, so returning a reference would return a dangling reference.

Both functions are also provided with the comparison criterion as an additional argument:

Click here to view code image

```
namespace std {
    template <typename T, typename Compare>
      const T& min (const T& a, const T& b, Compare cmp);
    template <typename T, typename Compare>
      T min (initializer_list<T> initlist, Compare cmp);
    ...
}
```

The comparison argument might be a function or a function object (see Section 6.10, page 233) that compares both arguments and returns whether the first is less than the second according to some particular sorting criterion.

The following example shows how to use the maximum function by passing a special comparison function as an argument:

Click here to view code image

```
// util/minmax1.cpp

#include <algorithm>

//function that compares two pointers by comparing the values to which they point
bool int_ptr_less (int* a, int* b)
{
    return *a < *b;
}

int main()
{
    int x = 17;
    int y = 42;
    int z = 33;
    int* px = &x;
    int* py = &y;
    int* pz = &z;

    //call max() with special comparison function
    int* pmax = std::max (px, py, int_ptr_less);

    //call minmax() for initializer list with special comparison function
    std::pair<int*,int*> extremes = std::minmax ({px, py, pz},
                                                 int_ptr_less);
    ...
}
```

Alternatively, you could use new language features, such as a lambda, to specify the comparison criterion and `auto` to avoid the explicit declaration of the return value:

Click here to view code image

```
auto extremes = std::minmax ({px, py, pz}, [](int*a, int*b) {
                                                 return *a < *b;
                                             });
```

Note that the definitions of `min()` and `max()` require that both types match. Thus, you can't call them for objects of different types:

```
int i;
long l;
...
std::max(i,l);              //ERROR: argument types don't match
std::max({i,l});            //ERROR: argument types don't match
```

However, you could qualify explicitly the type of your template arguments (and thus the return type):

```
std::max<long>(i,l);       //OK
std::max<long>({i,l});     //OK
```

5.5.2. Swapping Two Values

The function `swap()` is provided to swap the values of two objects. The general implementation of `swap()` is defined in `<utility>` as follows:[24]

[24] Before C++11, `swap()` was defined in `<algorithm>`.

```
namespace std {
    template <typename T>
    inline void swap(T& a, T& b)  ...  {
        T tmp(std::move(a));
        a = std::move(b);
        b = std::move(tmp);
    }
}
```

Thus, internally, the values are moved or move assigned if possible (see Section 3.1.5, page 19, for details of move semantics). Before C++11, the values were always assigned or copied.

By using this function, you can have two arbitrary variables `x` and `y` swap their values by calling

```
std::swap(x,y);
```

Of course, this call is possible only if move or copy semantics are provided by the parameter type.

Note that `swap()` provides an exception specification (that's why … is used in the previous declarations). The exception specification for the general `swap()` is:[25]

[25] See Section 3.1.7, page 24, for details about `noexcept` and Section 5.4.2, page 127, for the type traits used here.

```
noexcept(is_nothrow_move_constructible<T>::value &&
         is_nothrow_move_assignable<T>::value)
```

Since C++11, the C++ standard library also provides an overload for arrays:

```
namespace std {
  template <typename T, size_t N>
  void swap (T (&a)[N], T (&b)[N])
            noexcept(noexcept(swap(*a,*b)));
}
```

The big advantage of using `swap()` is that it enables you to provide special implementations for more complex types by using template specialization or function overloading. These special implementations might save time by swapping internal members rather than by assigning the objects. This is the case, for example, for all standard containers (see Section 7.1.2, page 258) and strings (see Section 13.2.8, page 674). For example, a `swap()` implementation for a simple container that has only an array and the number of elements as members could look like this:

```
class MyContainer {
  private:
    int* elems;         // dynamic array of elements
    int  numElems;     // number of elements
  public:
    ...
    // implementation of swap()
    void swap(MyContainer& x) {
        std::swap(elems,x.elems);
        std::swap(numElems,x.numElems);
    }
    ...
};

// overloaded global swap() for this type
inline void swap (MyContainer& c1, MyContainer& c2)
                 noexcept(noexcept(c1.swap(c2)))
{
    c1.swap(c2);      // calls implementation of swap()
}
```

So, calling `swap()` instead of swapping the values directly might result in substantial performance improvements. You should always offer a specialization of `swap()` for your own types if doing so has performance advantages.

Note that both types have to match:

Click here to view code image

```
int i;
long l;
std::swap(i,l);      // ERROR: argument types don't match
int a1[10];
int a3[11];
std::swap(a1,a3);    // ERROR: arrays have different types (different sizes)
```

5.5.3. Supplementary Comparison Operators

Four function templates define the comparison operators `!=`, `>`, `<=`, and `>=` by calling the operators `==` and `<`. These functions are declared in `<utility>` and are usually defined as follows:

Click here to view code image

```
namespace std {
    namespace rel_ops {
        template <typename T>
        inline bool operator!= (const T& x, const T& y) {
            return !(x == y);
        }
        template <typename T>
        inline bool operator> (const T& x, const T& y) {
            return y < x;
        }
        template <typename T>
        inline bool operator<= (const T& x, const T& y) {
            return !(y < x);
        }
        template <typename T>
        inline bool operator>= (const T& x, const T& y) {
            return !(x < y);
        }
    }
}
```

To use these functions, you need only define operators `<` and `==`. Using namespace `std::rel_ops` defines the other comparison operators automatically. For example:

Click here to view code image

```
#include <utility>

class X {
  public:
    bool operator== (const X& x) const;
    bool operator< (const X& x) const;
    …
};

void foo()
{
     using namespace std::rel_ops;   // make !=, >, etc., available
     X x1, x2;
     …
     if (x1 != x2) {    // OK
         …
     }
     if (x1 > x2) {    // OK
         …
     }
}
```

These operators are defined in a subnamespace of `std`, called `rel_ops`. They are in a separate namespace so that user-defined relational operators in the global namespace won't clash even if all identifiers of namespace `std` become global by using a general using directive:

```
using namespace std;          // operators are not in global scope
```

On the other hand, users who want to get their hands on them explicitly can implement the following without having to rely on lookup rules to find them implicitly:

```
using namespace std::rel_ops;          // operators are in global scope
```

Some implementations define the operators by using two different argument types:

Click here to view code image

```
namespace std {
    namespace rel_ops {
        template <typename T1, typename T2>
        inline bool operator!=(const T1& x, const T2& y) {
            return !(x == y);
        }
        …
    }
}
```

The advantage of such an implementation is that the types of the operands may differ, provided the types are comparable. But note that this kind of implementation is not provided by the C++ standard library. Thus, taking advantage of it makes code nonportable.