Part III

Beyond the STL: components and applications

Set operations on associative containers

6

Summary: *This chapter presents operations which are not included in the STL and which overcome the limitations described in Section 5.6.6. This has its price: these algorithms no longer work on simple C arrays and thus do not satisfy the requirements put on their algorithms by the authors of the STL. The price, however, is not too high, because algorithms and data structures should match. Thus, the data structures suitable for set operations are not necessarily sorted C arrays, but sets, represented, for example, by the* `set` *class.*

The algorithms in this chapter have a further advantage: they work not only on the sorted set containers of the STL, but also on unsorted associative containers as described in Chapter 7. Then, they are not slower than the set operations of Section 5.6. The algorithms of this chapter are not designed for multisets, but they can be extended accordingly.

The names of the algorithms differ from those of the STL because they lack the `set` prefix and have an upper case initial letter. All algorithms and examples in Part III, which starts with this chapter, are also available via the Internet (see page 271).

The `set_type` placeholder for the data type used in the following templates applies to all set containers that provide the following methods:

```
begin()
end()
find()
insert()
swap()
```

In addition, just the public type

```
set_type::const_iterator
```

must be available, by means of which elements of the set can be accessed. Obviously, the semantics of the methods and the iterator type must conform to the STL.

6.1 Subset relation

This algorithm determines whether a set `s2` is contained in a set `s1`. Each element of `s2` is checked to see whether it is included in `s1`:

```
// include/setalgo.h
#ifndef SETALGO_H
#define SETALGO_H
namespace br_stl {

template<class set_type>
bool Includes(const set_type& s1, const set_type& s2) {
    // Is s2 contained in s1?
    if(&s1 == &s2)        // save time if the sets are identical
       return true;
    /*The check for identity must not be confused with the check for equality which
      would have to be formulated as if(s1 == s2)...! The identity check is very
      fast, because only addresses are compared. The equality check can take a long
      time, because the sets must be compared element by element.
    */
    typename set_type::const_iterator i = s2.begin();

    while(i != s2.end()) {
        if(s1.find(*i++) == s1.end())      // not found
           return false;
    }
    return true;
}
```

The complexity is $O(N_2 \log N_1)$ for the STL class `set` and $O(N_2)$ for the class `HSet` in Chapter 7. Here and in the following sections, N_1 and N_2 denote the number of elements in `s1` and `s2`.

The check for identity of the arguments saves time because the loop is not executed. If `s2` is larger than `s1`, it cannot be contained in `s1` – a chance for further optimization (not shown in the code).

6.2 Union

This and the following algorithms have three sets as parameters, with the third parameter `result` containing the result after the end of the algorithm. When calling the function, `result` can be identical with `s1` or `s2`, so a temporary set is used to store the intermediate results. In order to save an assignment `result = temp`, which is expensive when many elements are involved, the member function `swap()` of the container is employed. `Union()` initializes `temp` with `s2` and adds all the elements of `s1`.

```
template<class set_type>
void Union(const set_type& s1, const set_type& s2,
           set_type& result) {
    set_type temp(s2);
    if(&s1 != &s2) {
       typename set_type::const_iterator i = s1.begin();
       while(i != s1.end())
            temp.insert(*i++);
    }
    temp.swap(result);
}
```

The `if` condition is used for speed optimization. If both sets are identical, there is no need for the loop. The complexity is $O(N_2 \log N_2 + N_1 \log N_1)$ for the STL class `set` and $O(N_2 + N_1)$ for the class `HSet` in Chapter 7. The first term of the sum refers to the initialization of `temp`, the second to the loop.

6.3 Intersection

The `Intersection()` algorithm begins with an empty container and inserts all the elements that are contained both in `s1` and in `s2`.

```
template<class set_type>
void Intersection(const set_type& s1, const set_type& s2,
                  set_type& result) {
    set_type temp;
    typename set_type::const_iterator i1 = s1.begin(), i2;

    // An identity check makes no sense, because in case
    // of identity, temp must be filled anyway.

    while(i1 != s1.end()) {
        i2 = s2.find(*i1++);
        if(i2 != s2.end())
           temp.insert(*i2);
    }
    temp.swap(result);
}
```

The complexity is $O(N_1 \log N_2)$ for the STL class `set` and $O(N_1)$ for the class `HSet` (Chapter 7). The factor N_1 refers to the loop, the rest to the `find()` operation. The function `insert()` is only called a maximum of $(\min(N_1, N_2))$ times and is therefore not considered in the complexity analysis.

Here too, a gain in speed could be achieved by running the loop on the smaller of the two sets.

6.4 Difference

Here, all the elements are inserted into `result` which are contained in `s1`, but not in `s2`.

```
template<class set_type>
void Difference(const set_type& s1, const set_type& s2,
                set_type& result) {
    set_type temp;
    typename set_type::const_iterator i = s1.begin();

    if(&s1 != &s2)
       while(i != s1.end()) {
           if(s2.find(*i) == s2.end())    // not found
               temp.insert(*i);
           ++i;
       }
    temp.swap(result);
}
```

The complexity is $O(N_1 \log(\max(N_1, N_2)))$ for the STL class `set` and $O(N_1)$ for the class `HSet` (Chapter 7). Calculation of the maximum is necessary, because for a small set `s2`, very many elements of `s1` must be inserted into `temp`, or for a large N_2, the number of `insert()` operations may also be small.

The check for non-identity (`&s1 != &s2`) saves the loop in case of identical arguments and immediately returns an empty set. Initializing of `temp` with `s1` and deletion of all elements contained in `s2` does not lead to a gain in time, because the possible savings in the loop are compensated by the cost of the initialization. Some time could, however, be saved by choosing the smaller set for the loop (see Exercise 6.1).

6.5 Symmetric difference

This algorithm finds all the elements that occur in `s1` or in `s2`, but not in both. The symmetric difference is equivalent to $(s1 - s2) \cup (s2 - s1)$ (implemented here) or $(s1 \cup s2) - (s1 \cap s2)$.

```
template<class set_type>
void Symmetric_Difference(const set_type& s1,
                          const set_type& s2,
                          set_type& result) {
    set_type temp;
    typename set_type::const_iterator i = s1.begin();

    if(&s1 != &s2) {
       while(i != s1.end()) {
           if(s2.find(*i) == s2.end())  // not found
              temp.insert(*i);
```

```
            ++i;
        }

        i = s2.begin();
        while(i != s2.end()) {
            if(s1.find(*i) == s1.end())   // not found
               temp.insert(*i);
            ++i;
        }
    }
    temp.swap(result);
}
} // namespace br_stl
#endif   // File setalgo.h
```

The complexity is $O((N_1 + N_2)\log(\max(N_1, N_2)))$ for the STL class `set` and $O(N_1 + N_2)$ for the class `HSet` (Chapter 7). The check for non-identity (`&s1 != &s2`) saves the loop in case of identical arguments and directly returns an empty set.

6.6 Example

This example contains a compiler switch `STL_set` which allows you to compile the program both with the set container of the STL and with the faster `HSet` container (Chapter 7). This shows the compatibility of the algorithms with two different set implementations. The switch controls not only the type definitions, but also the inclusion of a class `HashFun` used for the creation of a function object for the address calculation. `HashFun` serves as standard hash-function object, provided that no different object is required, and is stored in the file *hashfun.h*:

```
// include/hashfun.h
// Standard function object, see Chapter 7
#ifndef HASH_FUNCTION_H
#define HASH_FUNCTION_H

namespace br_stl {

template<class T>
class HashFun {
   public:
    HashFun(long prime=1009) : tabSize(prime) {}
    long operator()(T p) const {
       return long(p) % tabSize;
    }
    long tableSize() const { return tabSize;}

   private:
    long tabSize;
```

```
};
} // namespace br_stl
#endif
```

In order not to repeat the example in Chapter 7, it is recommended that you try it out again after reading the next chapter, commenting out the macro

```
// #define STL_set
```

This does not change the behavior of the program, only the underlying implementation – and with this, the running time.

```
// k6/mainset.cpp
// Example for sets with set algorithms
// alternatively for set (STL) or HSet(hash) implementation
#include<showseq.h>
#include<setalgo.h>

// compiler switch (see text)
#ifdef STL_SET
#include<set>
char msg[] = "std::set chosen";
#else
#include<hset.h>
#include<hashfun.h>
char msg[] = "br_stl::HSet chosen";
#endif

using namespace std;

int main() {
// type definition according to selected implementation
#ifdef STL_set
    // default setting for comparison: less<int>
    typedef set<int> SET;
#else
    typedef br_stl::HSet<int, br_stl::HashFun<int> > SET;
#endif

    SET  Set1, Set2, Result;
    int i;
    for(i = 0; i < 10; ++i) Set1.insert(i);
    for(i = 7; i < 16; ++i) Set2.insert(i);

    // display
    br_stl::showSequence(Set1);
    br_stl::showSequence(Set2);
    cout << "Subset:\n";
    cout << "Includes(Set1, Set2) = "
         << br_stl::Includes(Set1, Set2) << endl;
```

```
    cout << "Includes(Set1, Set1) = "
         << br_stl::Includes(Set1, Set1) << endl;

    cout << "Union:\n";
    br_stl::Union(Set1, Set2, Result);
    br_stl::showSequence(Result);

    cout << "Intersection:\n";
    br_stl::Intersection(Set1, Set2, Result);
    br_stl::showSequence(Result);

    cout << "Difference:\n";
    br_stl::Difference(Set1, Set2, Result);
    br_stl::showSequence(Result);

    cout << "Symmetric difference:\n";
    br_stl::Symmetric_Difference(Set1, Set2, Result);
    br_stl::showSequence(Result);

    cout << "Copy constructor:\n";
    SET newSet(Result);
    br_stl::showSequence(newSet);

    cout << "Assignment:\n";
    Result = Set1;
    br_stl::showSequence(Set1);
    br_stl::showSequence(Result);
}
```