# Fast associative containers

# 7

***Summary:*** *This chapter introduces associative containers which, because of hashing, allow significantly faster access times than the sorted associative containers of the STL. The chapter concludes with suitable overloaded operators for set operations on these containers.*

As already mentioned in Section 4.4, containers of this kind were not incorporated into the C++ standard for reasons of time, although the STL developers had made a corresponding proposal. Therefore, there is no standard for this kind of container. On the other hand, under certain conditions which will be explained below, access to the elements of these containers is independent of the number of elements, making it particularly fast ($O(1)$), so that containers of this kind are frequently employed.

This is reason enough to discuss hashing more extensively and, in particular, to present a solution based on the elements of the STL. Most compiler producers offer hash-based containers in their libraries; however they are not (yet) compatible with the STL. To make the underlying concepts as simply as possible, no reference is made to a vendor's special implementation. By omitting less important functions, such as the one for automatic adaptation to container size, the description can be made even clearer.

Applications, for example a sparse $1\,000\,000 \times 1\,000\,000$ matrix with fast access, will be shown in subsequent chapters. Matrices with large index ranges occur in simulations of networks (gas and electricity supplies, telecommunication, and so on). Compared with the sorted associative containers of the STL, the solutions presented are not only faster, but also more economical in their memory consumption.

## 7.1 Fundamentals

Sometimes, the sorting provided by the associative containers is not needed. The order of the elements of a set or map need not be defined. If we do not implement sorting we can calculate the address of a sought element directly from the key. For example, a compiler builds a symbol table whose elements must be able to be accessed very quickly. The complexity of the access is $O(1)$, independent of the number $N$ of elements in the table.

The preconditions are that the address can be calculated in constant time with a simple formula, that sufficient memory is available, and that the address calculation supplies an even distribution of elements in memory.

This kind of storage is called hashing. It is always suitable when the actual number of keys to be stored is small compared to the number of possible keys. A compiler can have a symbol table with 10 000 entries; the number of possible variable names with, for example, only 10 characters is much larger. If, for simplicity, we assume that only the 26 lower case letters are to be used, the result already shows $26^{10}$ = approx. $1.4 \cdot 10^{14}$ possibilities. The same problem arises with the storage of huge matrices in which only a small percentage of the elements is not equal to zero.

The function $h(k)$ for the transformation of the key $k$ into the address is called the *hash function* because all $N$ possibilities of keys must be mapped to $M$ storage places by hashing and mixing up information. $M$ is supposed to be very much smaller than $N$, which immediately creates a problem: two different keys may result in the same address. Such collisions must be taken into account. The function $h(k), 0 \leq k < N$ must yield only values between 0 and $M - 1$. A very simple hash function for numeric keys is the modulo function

$$h(k) = k \bmod M$$

Here, in order to achieve an even distribution, a prime number is chosen for the table size $M$. Nevertheless, the distribution strongly depends on the kind and occurrence of the keys, and it is sometimes difficult to find a function that leads to only a few collisions. A hash function for character strings should ensure that similar character strings do not lead to agglomerations in the hash table. The best way is to check the distribution by way of 'actual' data, in order to adapt the hash function appropriately before a software product is used.

### 7.1.1 Collision handling

What happens if two keys land on the same address? The second one comes off worse if the place is already occupied. One method is the so-called 'open addressing,' in which the attempt is made, by repeated application of the same (or another) hash function, to jump to a new, free address. This method assumes that there must be a tag for an address which shows whether it is free or not. When the table is filled, searching for and entering an element will take longer. Thus, the complexity $O(1)$ is an expectation value for a table which is not too full. Good addressing methods need approximately three or four calculations and associated jumps in order to find a free place with an occupation rate $\alpha$ of 90%. The occupation rate $\alpha$ is defined as the ratio of the number of entries and the size of the hash table.

Open addressing is problematic when elements are to be deleted, because the corresponding table entry cannot simply be marked as 'free.' It might well be that the entry has been previously used as a jumping point for finding the next address

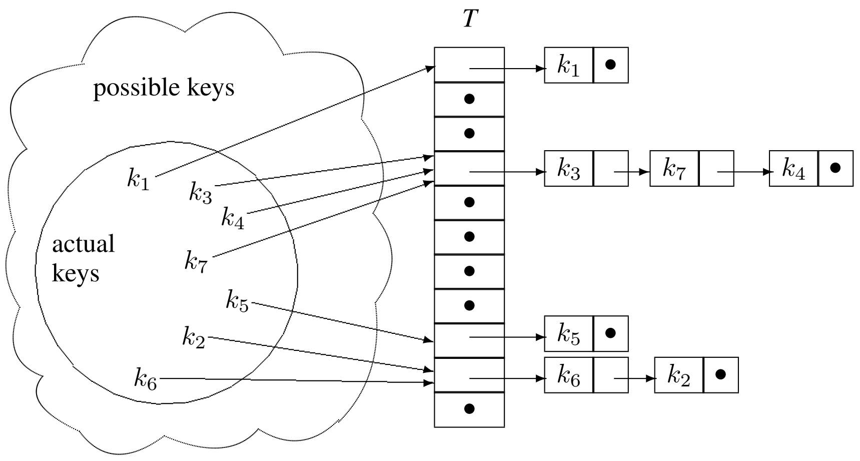

*Figure 7.1: Hashing with collision resolution by chaining.*

during insertion of another element. After the deletion, this other element can no longer be found.

Here is another common method in which the keys are not stored directly. Instead, each entry in the table consists of a reference to a singly-linked list in which all keys with the same hash function value are stored. This method is called 'hashing with collision resolution by chaining' and is shown in Figure 7.1.

A table element $T[i]$ points to a list of all keys whose hash function value equals $i$. In Figure 7.1 it holds that $h(k_1) = 0, h(k_3) = h(k_4) = h(k_7) = 3, h(k_5) = 8$ and $h(k_2) = h(k_6) = 9$. Deletion of an element is simpler, and because of the nearly unlimited length of a list, more elements can be stored than the table has positions. Such an occupation rate $> 1$ obviously entails a loss of performance, because in the worst case, the search or insertion time is proportional to the length of the longest list.

## 7.2 Map

This section gives a complete description of the hash-based class `HMap`, whose name differs from that of the STL class `map` by its upper case initial and a prefixed H. Under the assumption made on page 170, search or insertion of an element in `HMap` is carried out in constant time, that is, independently of the number $N$ of already existing elements, whereas in `map`, the same process is of complexity $O(\log N)$.

The internal data structure for the hash table is a vector `v` whose elements are pointers to singly-linked lists, as shown in Figure 7.1. The hash table $T$ of the figure is implemented by means of the vector `v`. A list is realized by means of the standard `list` class.

For reasons of simplicity and clearness, `HMap` implements only the most important type names and functions of the `map` class. However, the functions available in `HMap` have the same interface as `map`, so that all the following examples which do not assume sorting can work as well with `map`, only more slowly.

```
// File include/hmap.h (= hash map)
#ifndef HASHMAP_H
#define HASHMAP_H

// implicit data structures
#include<vector>
#include<list>

#include<cassert>
#include<algorithm>

namespace br_stl {

// hash map class
template<class Key, class T, class hashFun>
class HMap {
 public:
    typedef size_t size_type;
    typedef std::pair<const Key,T> value_type;

    // define more readable denominations
    typedef std::list<value_type> list_type;
    typedef std::vector<list_type*> vector_type;

    /*The template parameter Key stands for the type of the key; T stands for the class
      of data associated to a key; and hashFun is the placeholder for the data type
      of the function objects used for address calculation. Below, a function object for
      address calculation is proposed, but any other one can be used as well. Analogous
      to map, value_type is the type of the elements that are stored in a HMap object.
      value_type is a pair consisting of a constant key and the associated data.
    */

    class iterator;
    typedef iterator const_iterator; // maintain STL compatibility
    friend class iterator;

    /*The nested class iterator cooperates closely with HMap, so that both are mutu-
      ally declared as friend. iterator is only supposed to allow forward traversal
      and therefore its category is of the standard type forward_iterator_tag.
      An iterator object allows you to visit all the elements of a HMap object one after the
      other. Neither an order nor a sorting is defined for the elements. The visiting order
      of the iterator is given by the implicit data structure (see below, operator++()).
    */
    class iterator {
      friend class HMap<Key, T, hashFun>;
```

```
private:
 typename list_type::iterator current;
 typedef std::forward_iterator_tag iterator_category;
 size_type Address;
 const vector_type *pVec;
```

/* Privately, the `HMap` iterator must remember three things:

- `current`, an iterator for a list which begins at an element of the vector,
- `pVec`, a pointer to the vector on which the `HMap` iterator walks, and
- `Address`, the number of the vector element where the currently processed list begins.

The constructors initialize the private data, with the default constructor initializing the pointer to the vector with 0 and `current` implicitly with a list end iterator.
*/

```
public:
 iterator()
 : pVec(0) {
 }

 iterator(typename list_type::iterator LI,
          size_type A,  const vector_type *C)
 : current(LI), Address(A), pVec(C) {
 }
```

/*The following operators allow you to check a `HMap` iterator in the condition part of `if` or `while` as to whether it is at all defined:
*/

// type cast operator

```
 operator const void* () const { return pVec;}

 bool operator!() const { return pVec == 0;}
```

/*The operator for dereferencing occurs both in the `const` variation and in the non-`const` variation. Thus, dereferencing of an undefined iterator is punished with a program abort, which is a clear message to you to check the program that uses the iterator.
*/

```
 const value_type& operator*() const {
    assert(pVec);
    return *current;
 }
```

```
value_type& operator*() {
   assert(pVec);
   return *current;
}

/*The non-const variation is required to modify data independently of the key.
  Modification of the key must be excluded because it requires a new address
  calculation. Constancy is guaranteed by the const declaration in the type
  definition of value_type. How does the HMap iterator move from one el-
  ement to the other with operator++()? First, current is incremented:
*/

iterator& operator++() {
   ++current;

   /*If after this, current points to a list element, a reference to the iterator
     is returned (see below: return *this). Otherwise, the end of the list
     is reached.
   */
   if(current == (*pVec)[Address]->end()) {
      /*At this point, one address after the other is checked in the vector,
        until either a list entry is found or the end of the vector is reached.
        In the latter case, the iterator becomes invalid, because it can only
        move forward. In order to exclude further use, pVec is set to 0:
      */

      while(++Address < pVec->size())
         if((*pVec)[Address]) {
            current = (*pVec)[Address]->begin();
            break;
         }

      if(Address == pVec->size()) // end of vector reached
         pVec = 0;
   }
   return *this;
}

/*The postfix variation does not show any peculiarities. It remembers the old
  state in the variable temp, calls the prefix form, and returns the old state.
*/
iterator operator++(int) {
    iterator temp = *this;
    operator++();
    return temp;
}

/*The last two methods compare two HMap iterators. Two undefined or invali-
  dated iterators are always considered as equal:
*/
```

```
        bool operator==(const iterator& x) const {
            return pVec && x.pVec && current == x.current
                 || !pVec && !x.pVec;
        }

        bool operator!=(const iterator& x) const {
           return !operator==(x);
        }
    }; // iterator
```

/*With this, the nested class `iterator` is concluded, so that now the data and methods of the `HMap` class can follow:
*/

```
private:
    vector_type v;
    hashFun hf;
    size_type count;
```

/*`count` is the number of stored pairs of keys and data, `v` is the vector whose elements are pointers to linked lists, and `hf` is the function object used for calculation of the hash address.
*/

```
public:
    iterator begin() const {
        size_type adr = 0;
        while(adr < v.size()) {
           if(!v[adr])      // found nothing?
             ++adr;         // continue search
           else
             return iterator(v[adr]->begin(), adr, &v);
        }
        return iterator();
    }

    iterator end() const {
        return iterator();
    }
```

/*The method `begin()` supplies an iterator to the first element – provided it exists – in the `HMap` object. Otherwise, as with `end()`, an end iterator is returned. Iterators can become invalid if, after their generation, elements have been inserted or deleted in the `HMap` object.

The following `HMap` constructor needs a hash function object `f` as the parameter. If no function object is passed, a default object `f` is generated by means of the default constructor of the `hashFun` class. The vector is created in the suitable size `f.tableSize()`, and all elements are initialized with 0. It is assumed that the `hashFun` class provides the method `tableSize()` (see Section 7.2.1).
*/

```
HMap(hashFun f = hashFun())
: v(f.tableSize(),0), hf(f), count(0) {
}
/*What is meant by 'suitable size'? The hash table has a capacity P, with a prime
  number generally being chosen for P. On the other hand, the hash function object
  is used for address calculation; thus, this object too must know P. It is important
  that both function and vector denote the same P; therefore, a separate specification
  in the initialization of HMap and hash function object would be prone to errors.

  In order to avoid the hash function object having to procure the information on
  the capacity of the vector, the opposite method is followed: the vector is created
  in a size determined by the hash function object, assuming that the hash function
  object provides a method tableSize() for finding out the size of the table.
  This assumption is checked at compile time.
*/
HMap(const HMap& S) {
    hf = S.hf;
    // provide deep copy
    v = vector_type(S.v.size(),0);
    count = 0;
    // begin(), end(), insert(): see below
    iterator t = S.begin();
    while(t != S.end())
      insert(*t++);
}

~HMap()              { clear();}       // see below

HMap& operator=(const HMap& S) {
    if(this != &S) {
      HMap temp(S);
      swap(temp);  // see below
    }
    return *this;
}

/*clear() uses delete to call the destructor of each list referred to by a vector
  element. Subsequently, the vector element is marked as unoccupied.
*/
void clear() {
     for(size_t i = 0; i < v.size(); ++i)
        if(v[i]) {                     // does list exist?
             delete v[i];
             v[i] = 0;
        }
     count = 0;
}
```

```
/*In the following find() and insert() functions, the sought address within the
   vector v is calculated directly by means of the hash function object. If the vector
   element contains a pointer to a list, the list is searched in find() by means of
   the list iterator temp until either an element with the correct key is found or the
   list has been completely processed:
*/
iterator find(const Key& k) const {
    size_type address = hf(k);      // calculate address

    if(!v[address])
       return iterator();           // non-existent
    typename list_type::iterator
            temp =  v[address]->begin();

    // find k in the list
    while(temp != v[address]->end())
      if((*temp).first == k)
        return iterator(temp,address,&v); //found
      else ++temp;

    return iterator();
}

/*A map stores pairs of keys and associated data, where the first element (first) is
   the key and the second element (second) contains the data. find() returns an
   iterator which can be interpreted as a pointer to a pair. In order to obtain the data
   belonging to a key, the index operator can be called with the key as argument:
*/
T& operator[](const Key& k) {
     return (*find(k)).second;
}
/*If the key does not exist, that is, if find() returns an end iterator, a run time error
   occurs while dereferencing! (See the dereferencing operator on page 173.)

   The HMap class allows the insertion of an element only if an element with that key
   does not yet exist. If this is not desirable, it is easily possible to use HMap to build
   a MultiHMap class that allows multiple insertion of elements with identical keys.
   As in the STL, insert() returns a pair whose first part consists of the iterator
   that points to the found position. The second part indicates whether the insertion
   has taken place or not.
*/
std::pair<iterator, bool> insert(const value_type& P) {
    iterator temp = find(P.first);
    bool inserted = false;

    if(!temp) { // not present
        size_type address = hf(P.first);
        if(!v[address])
           v[address] = new list_type;
```

```
            v[address]->push_front(P);
            temp = find(P.first); // redefine temp
            inserted = true;
            ++count;
        }
        return std::make_pair(temp, inserted);
}
```

/*After the insertion, temp is redefined, because the iterator at first does not point to an existing element. The known auxiliary function make_pair() (page 20) generates a pair object to be returned.
*/

```
void erase(iterator q) {
```

/*If the iterator is defined at all, the member function erase() of the associated list is called. Subsequently, the list is deleted, provided it is now empty, and the vector element to which the list is attached is set to 0.
*/

```
    if(q.pVec) {            // defined?
        v[q.Address]->erase(q.current);

        if(v[q.Address]->empty()) {
            delete v[q.Address];
            v[q.Address] = 0;
        }
        --count;
    }
}
```

/*Sometimes, we would probably like to delete all the elements of a map that have a given key. In a HMap, this can at most be one element, but in a HMultimap, several elements might be affected.
*/

```
// suitable for HMap and HMultimap
size_type erase(const Key& k) {
    size_type deleted_elements = 0; // count

    // calculate address
    size_type address = hf(k);
    if(!v[address])
        return 0;           // not present

    typename list_type::iterator
         temp =  v[address]->begin();
```

/*In the following loop, the list is searched. A iterator called pos is used to remember the current position for the deletion itself.
*/

```
      while(temp != v[address]->end()) {
         if((*temp).first == k) {
            typename list_type::iterator pos = temp++;

            v[address]->erase(pos);
            // pos is now undefined

            --count;
            ++deleted_elements;
         }
         else ++temp;
      }

      /*The temporary iterator temp is advanced in both branches of the if instruc-
         tion. The operation ++ cannot be extracted in order to save the else, because
         temp would then be identical with pos which is undefined after the deletion,
         and a defined ++ operation would no longer be possible.
      */

      // delete hash table entry if needed
      if(v[address]->empty()) {
         delete v[address];
         v[address] = 0;
      }
      return deleted_elements;
   }

   /*Here are a couple of very simple methods. As opposed to other containers,
      max_size() does not indicate the maximum number of elements that can be
      stored in a HMap container, which is limited only by the capacity of the lists,
      but the number of available hash table entries. This information is more sensible,
      because the efficiency of a HMap depends on the occupation range α, assum-
      ing a good hash function. The occupation rate can easily be determined: α =
      size()/max_size().
   */

   size_type size()      const { return count;}
   size_type max_size()  const { return v.size();}
   bool empty()          const { return count == 0;}

   void swap(HMap& s) {
      v.swap(s.v);
      std::swap(count, s.count);
      std::swap(hf, s.hf);
   }
};

} // namespace br_stl

#endif    // File hmap.h
```

The `swap()` method swaps two `HMap` containers, using both the `swap()`-method of the vector container and an algorithm (see page 105) for swapping the remaining private data.

## 7.2.1 Example

The following example is taken from Section 4.4.3 and has been slightly modified. As in the example at the end of Chapter 6, the modification consists in the introduction of a compiler switch `STL_map` which allows you to compile the program both with the map container of the STL and with the `HMap` container just presented. The switch controls not only the type definitions, but also the inclusion of a class `HashFun` (file *hashfun.h*), used to create a function object for the address calculation.

```
// function object for hash address calculation
#ifndef HASH_FUNCTION_H
#define HASH_FUNCTION_H

namespace br_stl {
 template<class IndexType>
 class HashFun {
   public:
    // size of hash table: 1009 entries
    HashFun(long prime=1009)  // other prime numbers are possible
    : tabSize(prime) {
    }

    // simple hash function
    long operator()(IndexType p) const {
       return long(p) % tabSize;
    }

    // tableSize() is used by the constructor of a HMap
    // or HSet container for determination of the size
    long tableSize() const {
       return tabSize;
    }

   private:
    long tabSize;
 };
}
#endif

// k7/maph.cpp: Example for a map with hash map
#include<string>
#include<iostream>
```

```
// compiler switch (see text)
//#define STL_map
#ifdef STL_map
 #include<map>
               // comparison object: less<long>:
 typedef std::map<long, std::string> MapType;

#else
 #include<hmap.h>
 #include<hashfun.h>
 typedef br_stl::HMap<long, std::string,
                      br_stl::HashFun<long> > MapType;
#endif

typedef MapType::value_type ValuePair;

using namespace std;

int main() {
   // same as in k4/map1.cpp on page 80
}
```

The source code of the `main()` program remains unchanged with respect to page 80. However, the running program behaves differently when the compiler switch `STL_map` is not set and therefore a `HMap` container is used as underlying implementation: the output is *not* sorted.

## 7.3 Set

A set differs from a map in that the keys are also the data, that is, no separation exists. By changing the parts concerning pairs of keys and data, it is very easy to derive a corresponding `HSet` class from the `HMap` class of Section 7.2. Apart from pure name changes (`HMap` becomes `HSet`), these parts are so few that the `HSet` class is not listed here. Further modifications concern only the following points:

- Overloaded operators for set operations are added which will be discussed in the following sections.
- `HSet` has no index operator, because keys and data are the same.
- In the `HSet` class, the dereferencing operator for an iterator is present only in the `const` variation, since direct modification of an element must not be allowed because of the necessary address recalculation. In contrast, in the `HMap` class, the non-`const` variation is desirable for modifying data independently from their keys. As can be seen from the definition of `HMap::value_type`, constancy of the key is guaranteed.

Of course, `HSet` is included in the sample files available via the Internet (*hset.h*).

## 7.4 Overloaded operators for sets

Once you design a class for sets, it is only reasonable to provide the usual set operations as overloaded operators. In the STL, these operators do not exist for set containers, so an extension is presented here which is based on three design criteria:

- The choice of operator symbols partly orients with the symbols known from the Pascal programming language:
  - `+` for the union of two sets
  - `-` for the difference of two sets
  - `*` for the intersection of two sets

  For the symmetric difference, which corresponds to the exclusive or, the corresponding C++ operator `^` is chosen. The Pascal keyword `in` does not exist in C++ and it does not seem reasonable to choose another C++ symbol instead, so no operator for the subset relation is defined.

- The operators are implemented by means of the global set operations shown in Chapter 6.
- The binary operators `+`, `-`, `*`, and `^` make use of the short form operators `+=`, and so on.

The following description is based on the fact that all methods (i.e. short form operators) are defined inline in the class definition of `HSet` (file *hset*). The algorithms from *include/setalgo.h* (see chapter 6) are used. The corresponding binary operator is not a member function, i.e. defined outside the class definition.

### 7.4.1 Union

Exceptionally, no use is made of the function `Union()` of Chapter 6 to prevent creation of a copy of `*this`:

```
HSet& operator+=(const HSet& S) {       // Union
    // Union from include/setalgo.h is not used
    // in order to avoid generation of a copy of *this

    typename HSet::iterator i = S.begin();
    while(i != S.end()) insert(*i++);
    return *this;
}
```

```
// binary operator
template<class Key, class hashFun>
HSet<Key, hashFun> operator+(const HSet<Key, hashFun>& S1,
                             const HSet<Key, hashFun>& S2) {
```

```
     HSet<Key, hashFun> result = S1;
     return result += S2;
}
```

### 7.4.2 Intersection

```
HSet& operator*=(const HSet& S) {      // intersection
   Intersection(*this, S, *this);
   return *this;
}
```

```
// binary operator
template<class Key, class hashFun>
HSet<Key, hashFun> operator*(const HSet<Key, hashFun>& S1,
                             const HSet<Key, hashFun>& S2) {
     HSet<Key, hashFun> result = S1;
     return result *= S2;
}
```

### 7.4.3 Difference

```
HSet& operator-=(const HSet& S) {      // difference
   Difference(*this, S, *this);
   return *this;
}
```

```
// binary operator
template<class Key, class hashFun>
HSet<Key, hashFun> operator-(const HSet<Key, hashFun>& S1,
                             const HSet<Key, hashFun>& S2) {
     HSet<Key, hashFun> result = S1;
     return result -= S2;
}
```

### 7.4.4 Symmetric difference

```
HSet& operator^=(const HSet& S) {      // symmetric difference
   Symmetric_Difference(*this, S, *this);
   return *this;
}
```

```
// binary operator
template<class Key, class hashFun>
HSet<Key, hashFun> operator^(const HSet<Key, hashFun>& S1,
                             const HSet<Key, hashFun>& S2) {
     HSet<Key, hashFun> result = S1;
```

```
        return result ^= S2;
    }
```

## 7.4.5 Example

This example shows the application of the overloaded operators for set operations without giving a choice between the set implementation of the STL and a `HSet` container, because the former does not provide these operators.

```
// k7/mainseto.cpp
#include<showseq.h>
#include<hset.h>
#include<hashfun.h>
using namespace std;
using namespace br_stl;

int main() {
    typedef HSet<int, HashFun<int> > SET;
    SET  Set1, Set2, Result;

    for(int i = 0; i < 10; ++i) Set1.insert(i);
    for(int i = 7; i < 16; ++i) Set2.insert(i);
    showSequence(Set1);      // 0 1 2 3 4 5 6 7 8 9
    showSequence(Set2);      // 7 8 9 10 11 12 13 14 15

    cout << "Union: set += set\n";
    Result = Set1;
    Result += Set2;
    showSequence(Result);  // 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

    cout << "Intersection: set *= set\n";
    Result = Set1;
    Result *= Set2;
    showSequence(Result);  // 7 8 9

    cout << "Union: result = set + set\n";
    Result = Set1 + Set2;
    showSequence(Result);  // 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

    cout << "Intersection: result = set * set\n";
    Result = Set1 * Set2;
    showSequence(Result);  // 7 8 9

    cout << "Difference: result = set - set\n";
    Result = Set1 - Set2;
    showSequence(Result);  // 0 1 2 3 4 5 6

    cout << "Symmetric difference: result = set ^ set\n";
    Result = Set1 ^ Set2;
    showSequence(Result);  // 0 1 2 3 4 5 6 10 11 12 13 14 15
}
```