4 Standard Containers and Algorithms

Objective: to practice using the standard library container classes and algorithms, with emphasis on efficiency. To learn more about operator overloading and iterators.

Read:

- Book: containers and algorithms. Operator overloading, iterators.

1 Name Servers and the Container Classes

On the web, computers are identified by IP addresses (32- or 128-bit numbers). Humans identify computers by symbolic names. A name server is a component in the Domain Name System (DNS) that translates a symbolic name to the corresponding IP address. The DNS is a very large distributed database that contains billions (or at least many millions) of IP addresses and that receives billions of lookup requests every day. Furthermore, the database is continuously updated.

In this lab, you will implement a local name server in C++. With "local" we mean that the name server does not communicate with other name servers; it can only perform translations using its own database. The goal is to develop a time-efficient name server. You shall implement four versions of the name server, using different container classes. All four classes implement the interface `NameServerInterface`:

```
using HostName = std::string;
using IPAddress = unsigned int;
const IPAddress NON_EXISTING_ADDRESS = 0;

class NameServerInterface {
public:
    virtual ~NameServerInterface() = default;
    virtual void insert(const HostName&, const IPAddress&) = 0;
    virtual bool remove(const HostName&) = 0;
    virtual IPAddress lookup(const HostName&) const = 0;
};
```

`insert()` inserts a name/address pair into the database, without checking if the name already exists. `remove()` removes a name/address pair and returns `true` if the name exists; it does nothing and returns `false` if the name doesn't exist. `lookup()` returns the IP address for a specified name, or `NON_EXISTING_ADDRESS` if the name doesn't exist.

You shall use library containers and algorithms as much as possible. This means, for example, that you are not allowed to use any `for` or `while` statements in your solutions. (There is one exception: you may use a `for` or `while` statement in the hash function, see assignment A1d.)

A1. The definition of the class `NameServerInterface` is in the file *nameserverinterface.h*.

a) Implement a class `VNS` (vector name server) that uses an unsorted `vector` to store the name/address pairs. Use the `find_if` algorithm to search for a host name. The third parameter to the algorithm should be a lambda.

This implementation is clearly inefficient. A sorted vector would be a good alternative, but not for a name server with many insertions and deletions.

b) Implement a class `MNS` (map name server) that uses a `map` to store the name/address pairs. The average search time in this implementation will be considerably better than that for the vector implementation.

c) Implement a class `UMNS` (unordered map name server) that uses an `unordered_map` to store the name/address pairs.

d) An unordered map is implemented using a hash table. You shall compare this implementation with your own implementation of a hash table. Implement a class `HNS` (hash name server) that uses a hash table — a `vector` of `vector`'s — to store the name/address pairs.

The hash table implementation is open for experimentation: you must select an appropriate size for the hash table (given as an argument to the constructor) and a suitable hash function.[8,9] You should be able to obtain approximately the same search times as for the unordered map implementation.

Copy the makefile from one of the previous labs, modify it. Use the program *nstest.cc* to verify that the insert/remove/lookup functions work correctly. Then, use the program *nstime.cc* to measure and print the search times for the four different implementations, using the file *nameserverdata.txt* as input (the file contains 290,024 name/address pairs[10]).

A2. Examples of average search times in milliseconds for a name server with 290,024 names are in the following table.

	290,024	1,000,000
vector	0.667	
map	0.00096	
unordered	0.00036	
hash	0.00031	

Search the Internet for information about efficiency of searching in different data structures, or use your knowledge from the algorithms and data structures course, and fill in the blanks in the table. Write a similar table for your own implementation.

2 Bitsets, Subscripting, and Iterators

2.1 Bitsets

To manipulate individual bits in a word, C++ provides the bitwise operators `&` (and), `|` (or), `^` (exclusive or), and the shift operators `<<` (shift left) and `>>` (shift right). The standard class `bitset<N>` generalizes this notion and provides operations on sets of `N` bits indexed from `0` through `N-1`. `N` may be arbitrary large, so the bitset may occupy many words.

For historical reasons, `bitset` doesn't provide any iterators. We will develop a simplified version of the `bitset` class where all the bits fit in one word, and extend the class with iterators so it becomes possible to use the standard library algorithms with the class. Our goal is to provide enough functionality to make the following program work correctly:

```
int main() {
    // Define an empty bitset, set every third bit, print
    Bitset bs;
    for (size_t i = 0; i < bs.size(); i += 3) {
        bs[i] = true;
    }
```

[8] Note that a good hash function should take all (or at least many) of the characters of a string into account and that `"abc"` and `"cba"` should have different hash codes. For instance, a hash function that merely adds the first and last characters of a string is not acceptable.

[9] `std::hash<string>` is a good hash function.

[10] The computer names are from `http://httparchive.org`. The IP addresses are running numbers.

```
    copy(bs.begin(), bs.end(), ostream_iterator<bool>(cout));
    cout << endl;

    // Find the first five bits that are set, complement them, print
    size_t cleared = 0;
    auto it = bs.begin();
    while (it != bs.end() && cleared != 5) {
        it = find(it, bs.end(), true);
        if (it != bs.end()) {
            *it = !*it;
            ++cleared;
            ++it;
        }
    }
    copy(bs.begin(), bs.end(), ostream_iterator<bool>(cout));
    cout << endl;

    // Count the number of set bits, print
    cout << "Number of set bits: " << count(bs.begin(), bs.end(), true) << endl;
}
```

The output from the program should be (on a 64-bit computer):

```
1001001001001001001001001001001001001001001001001001001001001001
0000000000000001001001001001001001001001001001001001001001001001
Number of set bits: 17
```

An iterator for bitsets has to support both reading and writing, so it must be a model of `ForwardIterator`. Actually, it is not difficult to make it a model of `RandomAccessIterator`, but this would mean that we had to supply more functions.

The solution will be developed in several steps:

- Implement the "bit fiddling" methods necessary to set, clear, and test an individual bit in a word (this we have done for you).
- Implement `operator[]`. This is rather difficult.
- Implement the bitset iterator. This turns out to be relatively simple.

A3. The files *simplebitset.h* and *simplebitset.cc* contain the implementation of a simple version of the bitset class, with `get` and `set` functions instead of a subscripting operator. Study the class and convince yourself that you understand how the bits are manipulated. Copy the makefile from one of the previous labs, modify it. Use the program in *simplebitsettest.cc* to check the function of the class.

2.2 Subscripting

Subscripting is handled by `operator[]`. In order to allow subscripting to be used on the left hand side of an assignment operator, `operator[]` must return a reference (e.g., like `int& operator[](int i)` in a `vector` class). For a bitset, a reference to an individual bit in a word is needed, but there are no "pointers to bits" in C++. We must write a "proxy class", `BitReference`, to represent the reference. This class contains a pointer to the word that contains the bits and an integer that is the position of the bit in the word.

Outline of the class (`BitsetStorage` is the type of the word that contains the bits):

```
class BitReference {
public:
    BitReference(Bitset::BitStorage* pb, std::size_t p) : p_bits(pb), pos(p) {}
    // ... operations will be added later
private:
    Bitset::BitStorage* p_bits; // pointer to the word containing bits
    std::size_t pos;            // position of the bit in the word
};
```

The `Bitset` class looks like this:

```
class Bitset {
    friend class BitReference;
public:
    ...
    bool operator[](std::size_t pos) const;
    BitReference operator[](std::size_t pos);
    ...
private:
    using BitStorage = unsigned long;
    BitStorage bits;
    static const std::size_t
        BPW = std::numeric_limits<BitStorage>::digits; // "Bits per word"
};
```

The const version of `operator[]` is easy: it is identical to the `get` function in `SimpleBitset`. The non-const version should be defined as follows:

```
BitReference operator[](std::size_t pos) {
    return BitReference(&bits, pos);
}
```

The actual bit fiddling is performed in the `BitReference` class. In order to see what we need to implement in this class we study the results of expressions involving `operator[]`:

```
bs[3] = true; // bs.operator[](3) = true; =>
              // BitReference(&bs.bits,3) = true; =>
              // BitReference(&bs.bits,3).operator=(true);
```

From this follows that the following operator function must be implemented in `BitReference`:

```
BitReference& operator=(bool b); // for bs[i] = b
```

This function should set the bit referenced by the `BitReference` object to the value of `b` (just like the `set` function in the `SimpleBitset` class). There are more ways of using `operator[]`:

```
bool b = bs[6]; // b = bs.operator[](6); =>
                // b = BitReference(&bs.bits,6); =>
                // b = BitReference(&bs.bits,6).operator bool();
```

A conversion function must be implemented:

```
operator bool() const; // for b = bs[i]
```

The last use case:

```
bs[3] = bs[6];  // bs.operator[](3) = bs.operator[](6); =>
                // BitReference(&bs.bits,3) = BitReference(&bs.bits,6); =>
                // BitReference(&bs.bits,3).operator=(BitReference(&bs.bits,6));
```

Another assignment operator must be implemented:

```
BitReference& operator=(const BitReference& rhs); // for bs[i] = bs[j]
```

A4. Use the files *bitset.h*, *bitset.cc*, *bitreference.h*, *bitreference.cc*, and *bitsettest1.cc*. Implement the functions in *bitreference.cc* and test.

2.3 Iterators

From one of the OH slides: "An iterator "points" to a value. All iterators are `DefaultConstructible` and `Assignable` and support `++it` and `it++`." A `ForwardIterator` should additionally be `EqualityComparable` and support `*it` for both reading and writing via the iterator.

The most important requirement is that an iterator should point to a value. A `BitsetIterator` should point to a Boolean value, and we already have something that does this: the class `BitReference`. The additional requirements (`++`, equality test, and `*`) are easy to implement in the iterator class. It will look like this:[11]

```
class BitsetIterator : public std::iterator<std::forward_iterator_tag, bool> {
public:
    BitsetIterator(Bitset::BitsetStorage& pb, size_t p) : ref(pb, p) {}
    bool operator!=(const BitsetIterator& bsi) const { ... }
    BitsetIterator& operator++() { ... }
    BitReference operator*() { ... }
    BitsetIterator& operator=(const BitsetIterator& rhs) {
        ref.p_bits = rhs.ref.p_bits;
        ref.pos = rhs.ref.pos;
        return *this;
    }
private:
    BitReference ref;
};
```

The base class `iterator` contains some type aliases, for example `value_type`, and the iterator tag `forward_iterator_tag`, which informs the compiler that the iterator is a forward iterator. The assignment operator is redefined so it makes a memberwise copy of the `BitReference` object, rather than using the assignment operator in `BitReference` which sets a bit in the bitset.

A5. Uncomment the lines in *bitreference.h*, *bitset.h* and *bitset.cc* that have to do with iterators, implement the `begin()` and `end()` functions. Implement the member functions in *bitsetiterator.h*. Use the program *bitsettest2.cc* to test your classes.

[11] This class only contains the constructs that are necessary for the test program. For example, we have not implemented postfix `++`, `->` or comparison with `==`.