CPP Certified Professional Programmer

LaTeX2e

Advanced CPP programming language

CPP institute

CPP institute
CPP institute

This dissertation is submitted for the degree of
Certified course

Physics Simulation

December 2020

Table of contents

List of figures

List of tables

Chapter 1

STL sequential containers

1.1 Quick introduction to Standard Template Library.

1.1.1 Basic information about STL

Note: This course is in BETA version. Even though we do our best effort to ensure the courseware is of very good quality, some occasional errors, spelling mistakes, and formatting issues might occur both in the learning resources and assessments. We apologize for all of them. The course will be fully updated in the first quarter of 2018, most probably in February. Thank you for your patience and being understanding.

STL stands for **Standard Template Library**. STL is an important part of the C++ development environment, and if you intend to be a C++ programmer, you should certainly learn how to use it.

What is STL good for? Generally speaking, it provides ready-to-use solutions for many programming problems. The word "standard" is also meaningful. It means that every C++ compiler is accompanied by **this software package**. So, as long as you use STL, your program portability is high (portability of C++ programs is a huge but unrelated topic). What problems can we solve with the help of the STL? Many, but not all – otherwise **we wouldn't need Boost libraries (free and peer-reviewed portable C++ source libraries)** and all of the language improvements.

Basically, STL can be divided into two parts:

- **Containers**, and
- **Algorithms**

There are more features in the library, and we'll explain them later in the course – don't worry. We just think that **containers and algorithms are most important**, and that's why we're going to focus on them first and most extensively.

1.1.2 Quick introduction to containers

Containers, or **collections**, as they are sometimes called, are meant to contain something inside. This something is various kinds of data. Different types of containers store data in different manners. This is good, because it allows programmers to choose the best collection for the task they face. We'll talk more about this later. The **simplest container** you can think of is an **array**, for example:

```
int a[10];
```

The array above is of type `int`. **An array is a container which stores elements of the same type**, and usually **those elements occupy a continuous area of memory**. Each element is placed one after the other, starting from a certain location in the memory. You can access a particular element of an array by its index using the `[] operator` – the square brackets operator.

For example:

```
a[3] = 3;
cout<<a[3]<<endl;
```

An array is a very basic container. It has a few **limitations**:

- the size of an array cannot be changed once instantiated;
- an array does not know its size – this information has to be stored in another variable;
- as a result of the previous point, an array also cannot check if it is properly accessed – the index used in the operator cannot be checked;
- an array (one dimension) is organized as one big memory block.

The last point is not always a limitation; it all depends on the particular array usage scenario.

1.1.3 Improving arrays

Let us think for a while about how to improve our array and remove some of **its limitations**. What could you do?

Let's design a `class` named `Array`, which will be our new better array. This `class` will remove the first two limitations of the original array: **the inability to change the size**, and **the lack of knowledge of its size (may we can know the size by `sizeoff()`)**. The class in its basic form might look like the one on the slide.

As you can see, the `Array` class is built around a regular C/C++ dynamically allocated array. But there's more. We've used class abstraction to improve the ordinary array. On the next slide, the newly created class will be put to use.

File name: 1.1.3.h

```
#ifndef ARRAY_H
#define ARRAY_H
#include <exception>
#include <iostream>

namespace CPP_course {
  class Array {
    int * _array;
    unsigned int _size;
  public:
    Array(unsigned size = 0);
    void add(int value);
    void delItem(unsigned index);
    virtual ~Array();
    unsigned int getSize() const;
```

```
    int & operator[](unsigned index);
  };
}
#endif
```

File name: 1.1.3.cpp

```
#include "1.1.3.h"

namespace CPP_course {
  Array::Array(unsigned size) : _array(0), _size(size) {
    if (size > 0){
      _array = new int[this->_size];
    }
  }

  void Array::add(int value){
    if (_size == 0){
      _array = new int[1];
    }
    else{
      int * tmp = new int[_size + 1];
      for (unsigned i = 0; i < _size; i++){
        tmp[i] = _array[i];
      }
      delete[] _array;
      _array = tmp;
    }
    _array[_size++] = value;
  }

  void Array::delItem(unsigned index){
    if (_size == 1){
      delete[] _array;
      _array = 0;
    }
    else{
      int * tmp = new int[_size - 1];
      for (unsigned i = 0, j = 0; i < _size; i++, j++){
        if (i == index){
          j--;
          continue;
        }
        tmp[j] = _array[i];
      }
```

```
      delete[] _array;
      _array = tmp;
    }
    _size--;
  }

  unsigned int Array::getSize() const {
    return _size;
  }

  int & Array::operator [](unsigned index){
    if (index > _size - 1){
      std::exception e;
      throw e;
    }
    return _array[index];
  }

  Array::~Array(){
    delete[] _array;
  }
}
```

1.1.4 Using our improved array

The first step is to declare an object of type `Array`. Its initial size is set to 10. The constructor allocates an array of ten elements inside object `A`.

The second interesting thing is that we use this object in the same way as a regular array `operator[]` is called for access. Such behavior is possible because the class implements `operator[]`.

The loops illustrated in the code are using the `getSize()` method to obtain the actual size of the Array object. This is another improvement over a standard C++ array.

File name: use1.1.3.cpp

```
#include "Array.h"

int main(){
  CPP_course::Array A(10);
  for (unsigned i = 0; i < A.getSize(); ++i){
    A[i] = i;
  }
  for (unsigned i = 0; i < A.getSize(); ++i){
    std::cout << A[i] << " ";
  }

  std::cout << "\n";
```

```
  return 0;
}
```

Let's go further. You've probably noticed two methods which have not been discussed yet: `add` and `delItem()`. Now we're going to put them to work.

Take a look at the example on the slide. This is the `main()` function, which we've seen before, with additional code. The code takes advantage of the methods `add` and `delItem`. These two methods allow our new array to grow and shrink as needed. Another array limitation seems to have been overcome. The last important modification is related to access control, which is provided inside `operator[]`. In case a program tries to access an element by an index which is out of scope, an exception is thrown.

1.1.5 Better array – STL way

Of course, the `Array` class is far from perfect. There's a lot of other functionality that could be implemented here. First of all, the class could be created as a template class. It would allow other types than just integer. Secondly, some performance issues are present in the previous code. But this is just a brief introduction to the world of containers. Now let's take a look at how STL fits into this picture. In order to do that, we'll rewrite the `main()` function using a very basic STL container – a vector. As you can see, the `vector` class allows for the same behavior as our handmade `Array` class. Moreover, the `vector` is a template class and can work with basically **any type of element**. That is a **huge advantage**.

In this example, we're using two very common `vector` methods: `push_back()` – which adds a new element to the end of a vector, and `pop_back()` – this one removes the last element from a vector. As the `Array` class, the vector also has `operator[]` overloaded. And finally, the method `size()`, whose task is rather obvious. This is a vector, in a nutshell, ladies and gentlemen. Let's see what else can be found inside the **Standard Template Library**. With the introduction of C++11, we can use a very compact way to initialize vector elements – the initializer list. We'll present it here shortly; it's just a comma-separated list enclosed in braces: `vector<int> v1 = 5, 6, 9;`

File name: 1.1.5.cpp

```
#include <vector>
#include <iostream>

using namespace std;

int main()
{
  vector <int> v1(10);
  vector <int> v2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  for(unsigned i = 0; i < v1.size(); ++i)
  {
          v1[i] = i;
  }
  for(unsigned i = 0; i < v1.size(); ++i)
  {
          cout << v1[i] << " ";
          cout << v2[i] << " ";
```

```
  }
  cout << endl;

  cout << v1.size() << endl;
  v1.push_back(100);
  cout << v1.size() << endl;
  v1.pop_back();
  cout << v1.size() << endl;
  return 0;
}
```

1.1.6 Standard Template Library

The STL can be divided into the following parts:

- containers
- algorithms
- input/output
- strings
- numeric library
- iterators
- utilities
- localizations
- regular expressions
- atomic operations
- thread support
- concepts

This division is not meant to be definitive. Different books or websites might provide different views of the STL. Let us take a closer look at each of the first seven categories.

1.1.7 Containers

This is one of two major parts of the STL. It is made up of various containers which help the programmer to implement the best solution to the task at hand.

The **Standard Template Library** provides the programmer with a lot of different containers, split into the following subcategories:

- sequential containers:

  - vector
  - list
  - deque
- associative containers:
  - set
  - multiset
  - map
  - multimap
- container adaptors:
  - stack
  - queue
  - priority queue

If you're familiar with abstract data structures, some of these names will be quite familiar – list, set, map, stack, etc. Containers are based on some underlying data abstraction, which is usually a well-known computer science concept. All this will be explained later.

1.1.8 Algorithms

The second biggest STL branch is **algorithms** – a means to transform data – usually stored inside either a particular or a general container. **These algorithms are expressed as functions**. There's a huge number of them, and if you're thinking about implementing an algorithm, it would be prudent to first check if it's not already present in the STL. Once again, we need to emphasize **the importance of using a well-defined library**, instead of creating your own versions. **Containers are structures only. Without algorithms, they're not of very much use**. Different sources divide STL algorithms into different categories. Below you can find a typical list:

- non-modifying sequence operations
- modifying sequence operations
- sorting
- set operations
- binary search
- heap operations
- min/max operations

The list is quite impressive; it's highly probable that you'll find what you are looking for here.

1.1.9 Input and output library

This part of the STL is responsible for all kinds of input and output operations: **console and files**. You certainly know the `cout` object and the `iostream` include. This is exactly the STL i/o division.

1.1.10 String library

The string library is C++'s response to the char * problem. The class string is a solution to many problems related to the processing of character strings. The class is not perfect, especially if you're familiar with its Java counterpart. There are also some performance considerations, but still, this class solves a lot of common string problems. Also, it's worth mentioning that the string class is a container itself, and can be treated like one.

1.1.11 Numeric library

A few additional classes are strongly connected to the math world. There is **the complex class**, which is exactly what you would expect, and the valarray type. **Valarray** is a special kind of array which allows for specific mathematical operations, like slicing.

1.1.12 Iterators

Iterators are generalizations of **pointers**. They allow for access to the elements of collections. Every collection, regardless of its type, can be accessed using iterators. There are five types of iterators to distinguish:

- input iterator
- output iterator
- forward iterator
- bidirectional iterator
- random access iterator

C++ iterators are not defined in any manner of specific type (like a class). Instead, they're defined by their behavior-supported operations. Every type of iterator has its own set of operations, which must be supported for a particular type to be called an iterator. This is the only constraint.

1.1.13 Utilities

The last department of **the Standard Template Library** is called **Utilities**. As the name suggests, it contains some **tools**, for example, for **date/time manipulation**, some supporting types like **pair**. The most important part is **the functional objects** (and functions) library. These objects are simple algorithms, defined for easy common tasks. For example, less(), which is used during the usage of the sort algorithm (and many more cases). The functional object library was designed to complement containers and algorithms in STL branches, but most of them can also be used in other applications.

1.2 Sequence Containers

1.2.1 Sequence Containers

These containers maintain a certain order to the elements inside them. This order can be completely controlled by a programmer, and he or she can choose the exact position at which a particular element will be located/placed in a sequence container. The containers themselves have no influence on the sequence of elements.

In this category, the STL offers three solutions:

- vector
- deque
- list

Before we describe the containers in detail, let's take a look at the table, which shows the various methods provided by each of them. As you can see, most of the methods are common among all three containers. But don't worry, each of the methods will be described appropriately, and with an accompanying example.

1.2.2 `Vectors`

Header: `<vector>`
Definition: `template<class T, class Allocator = std::allocator<T>> class vector;`

The vector is usually the first container to learn when somebody starts his or her adventure with the STL. A vector is basically a dynamic array. It always occupies a continuous memory block. The example at the start of this tutorial shows how a dynamic array works. The biggest advantage of the vector is the ability to access its elements randomly (by means of the index) in constant time.

The `allocator` is responsible for providing a **memory model** for the container elements. Usually the default one is used (`std::allocator<T>`), but it's also possible to supply a different one. Technically, you're allowed to use a different `allocator` for each type of container.

The definition of a vector shows that a vector is a template class and **it's necessary to specify the type of its elements during instantiation**. For example:

```
class A;
vector<int> v1;
vector<float> v2;
vector<A> v3;
```

1.2.3 Constructing `vectors`

Before we can start using the `vector`, we need to create it. We already know that we need to choose the type of elements of the `vector`, but this isn't the only choice we have to make. The `vector` class possesses a few constructor methods, which can be used as needed:

```
explicit 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 );
```

Default constructor

The first constructor is the default constructor. We've seen its usage before. The `allocator` parameter is optional in all the constructors.

Initializing the constructor

The second constructor creates a `allocator` containing n objects, all of them having the same value – value. It can be used in situations where someone needs a vector of a predefined size and wants to initialize it. Look at the example.

File name: 1.2.4.cpp

```
#include <vector>
#include <iostream>

using namespace std;

int main(){
  vector<int> v1(10,0);
  for (unsigned i = 0; i < v1.size(); ++i){
    v1[i] = i+1;
  }
  cout << "Size (v1): " << v1.size() << endl;
  for (unsigned i = 0; i < v1.size(); i++){
    cout << v1[i] << " ";
  }
  cout << endl;

  vector<int> v2(v1.begin(), v1.begin()+5);
  cout << "Size (v2): " << v2.size() << endl;
  for (unsigned i = 0; i < v2.size(); ++i){
    cout << v2[i] << " ";
  }
  cout << endl;
  return 0;
}
```

1.2.4 Constructing vectors Iterator constructors

The next constructor uses `iterators` to initialize itself. It will create a `vector` with a copy of the values from first (inclusive) to last (exclusive). In the most typical case, this constructor creates a new `vector` using the elements from an already existing collection.

But due to the fact that `iterators` are defined as a set of operations instead of a certain type, it is also possible to use normal pointers. This remark leads us to the conclusion that you can use a normal C++ array to initialize a collection. And in fact, you can.

1.2.5 Iterator constructors – initialization

As you can see, you can initialize a vector using a range of elements from another vector (or a collection in general) or an array. We need to remember again that the first element will be included, and the last element will not be in that range.

File name: 1.2.5.cpp

```
#include <vector>
#include <iostream>

using namespace std;

int main(){
  int a1[] = {1,2,3,4,5,6,7,8,9,10};

  vector<int> v1(a1, a1+10);
  cout << "Size (v1): " << v1.size() << endl;
  for (unsigned i = 0; i < v1.size(); ++i){
    cout << v1[i] << " ";
  }
  cout << endl;

  vector<int> v2(a1+5, a1+10);
  cout << "Size (v2): " << v2.size() << endl;
  for (unsigned i = 0; i < v2.size(); ++i){
    cout << v2[i] << " ";
  }
  cout << endl;
  return 0;
}
```

1.2.6 Copy constructors

The copy constructor is a very important concept in the C++ world, so it's not surprising that a vector also possesses one. It works in an obvious way, creating a new object from the existing one.

The copy is exact; therefore, every element from the source is copied. It's a very important fact, so remember this for now. On this page, there is an example of a copy constructor. Nothing fancy, but it shows the idea how the copy constructor is supposed to work.

On the next page, we'll show you a more advanced example.

File name: 1.2.6.cpp

```
#include <vector>
#include <iostream>

using namespace std;

int main(){
  int a1[] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v1(a1, a1+10);
  cout << "Size (v1): " << v1.size() << endl;
  for (unsigned i = 0; i < v1.size(); ++i){
    cout << v1[i] << " ";
  }
  cout << endl;

  vector<int> v2(v1);
  cout << "Size (v2): " << v2.size() << endl;
  for (unsigned i = 0; i < v2.size(); ++i){
    cout << v2[i] << " ";
  }
  cout << endl;
  return 0;
}
```

1.2.7 Dealing with objects

In all the examples presented so far, the `vectors` have been defined as of type `int`. Now we'll have to take a look at what happens when the type is not a C++ built-in one. There are many things that have to be taken into consideration.

Let's go through all the constructors again, but this time, let's create a `vector` of the objects. We need to pay attention to the behavior of the constructors (normal, default and copy as well) and assignment operators. Also, some aspects of type conversion will come into the picture.

Take a look at the example. The topic is: **`implicit` type conversion**. What do you think of this example? Will it compile or not? The answer is yes, it will. There is this constructor inside the class, which takes the int parameter.

The `push_back()` method expects an object of type `A`. This constructor will be used to create an object of type `A implicitly`. This can be an advantage, but it might also pose a danger if that constructor does something unwanted.

For example, an object might be created `implicitly` and passed to a method when we do not want it to happen in such a way. This would not be allowed if the constructor was marked `explicit`. Let's see this on the next page.

File name: 1.2.7.cpp

```
#include <vector>
#include <iostream>
```

```
using namespace std;

class A {
  int number;
public:
  A(int _number) : number(_number) {}
};

int main() {
  vector<A> v1;
  v1.push_back(1);
  return 0;
}
```

1.2.8 Constructors and objects

Let's move on further. The purpose of this example is to show you explicitly when a particular type of constructor is being called. It's not always obvious, therefore you might forget to create a constructor.

The numbers in brackets `()` indicate a particular line of code and the corresponding lines of output. Let's try to analyze the code and its behavior.

The first line creates a `vector` of size one. We've used the initializing constructor. Its first argument is a number of elements, and the second one is a possible value to initialize the elements. If the value is not submitted, the default value is used. It that case, an object of class `A` is created using a default constructor. The object is then passed to the constructor to initialize one element of the collection.

In the second line, an object is created using the type conversion mechanism we described earlier. The one-parameter constructor is invoked. The next step happens inside the `vector v1`. First it reallocates itself to make room for another object. Then two objects – the new one and the old one – are copied into the `vector` storage area.

The third line is just an assignment. But again, the first step is to create an object, then to assign it to its proper place.

This example clearly shows how important it is to have proper constructors, the copy constructor and the assignment operator in a class which is meant to be put into STL containers. You won't always need to create all these methods explicitly. On many occasions, it'll be sufficient to rely on their default implementations.

File name: 1.2.8.cpp

```
#include <vector>
#include <iostream>

using namespace std;

class A {
  int number;
public:
  explicit A(int _number) : number(_number) {}
};
```

```
int main() {
  vector <A> v1;
  v1.push_back(1);
  return 0;
}
```

1.2.9 Creating a copy of a vector

The final example will show what is happening while copying the entire `vector`. We're using the same class `A` as in the previous example.

Let's carry out a short analysis of what's going on in this code. Again, the lines of code and the corresponding output are marked.

Part one is easy: the `v1 vector` is created and one element is added to it.

Part two: the copy constructor is invoked. Inside the source `vector` there is only one element, so the class `A` copy constructor is called only once.

Part three: this one is a little bit surprising. The output clearly states that the copy constructor has been called, but we were expecting the assignment operator instead, were we not? The target vector is empty. It looks as though in such a case, the copy constructor is called instead of the assignment operator. This situation is worth remembering.

Part four – the last one: this is pretty straightforward. A `vector` containing two objects of class A is created, so there are two invocations of the default constructor and the copy constructor. The next step is an assignment operation, and since the source vector has only one element, only one invocation of the assignment operator is performed.

File name: 1.2.9.cpp

```
#include <vector>
#include <iostream>

using namespace std;

class A {
  int number1;
  int number2;
public:
  A(int _number) : number1(_number), number2(0) {
    cout << "Normal constructor\n";
  }

  A() {
    cout << "Default constructor\n";
  }

  explicit A(const A& source) {
    number1 = source.number1;
```

```
    number2 = source.number2;
    cout << "Copy constructor\n";
  }

  A & operator=(const A& source) {
    number1 = source.number1;
    number2 = source.number2;
    cout << "Assignment operator\n";
    return *this;
  }
};

int main() {
  vector <A> v1(1);
  cout << "\n";
  v1.push_back(1);
  cout << "\n";
  v1[0] = 10;
  cout << v1[0].number1 << " " << v1[0].number2 << "\n";
  cout << "\n";
  vector <A> v2(v1);

  return 0;
}
```

1.2.10 Destructors

There's not much to say about this method. It does the housework. It destroys all the objects stored inside a `vector` by calling their destructors, and then deallocates all storage capacity reserved by the vector.

But it's important to remember that the destructor of elements will be called if and only if these elements are static (not pointers). In the opposite case, the user is responsible for this activity.

Take a look at the example on the slide. A lot has happened in this code. We should focus on the output after the Third ready! mark. We can see three invocations of the destructor. All of them have been caused by the destructor of the `vector`.

File name: 1.2.10.cpp

```
#include <vector>
#include <iostream>

using namespace std;

class A {
  int number1;
  int number2;
public:
```

```
  A(int _number) : number1(_number), number2(0) {
    cout << "Normal constructor\n";
  }

  A() {
    cout << "Default constructor\n";
  }

  explicit A(const A& source) {
    number1 = source.number1;
    number2 = source.number2;
    cout << "Copy constructor\n";
  }

  A & operator=(const A& source) {
    number1 = source.number1;
    number2 = source.number2;
    cout << "Assignment operator\n";
    return *this;
  }
};

int main() {
  vector <A> v1;
  v1.push_back(1);
  cout << "First ready!\n";
  vector <A> v2(v1);
  cout << "Second ready!\n";
  vector <A> v3;
  v3 = v2;
  vector <A> v4(2);
  v4 = v2;
  return 0;
}
```

1.2.11 Destroying dynamically allocated objects

Now, for comparison, let's see what happens when dynamically allocated objects are in the collection. As you can see, there are no destructors calls whatsoever. Dynamically allocated objects stored inside a collection will not be automatically destroyed when this collection is being destroyed. This can lead to a memory leak, which is already bad enough. But there are scenarios where unwanted objects can lead to more severe consequences.

Although this example involves the vector only, it works in exactly the same manner for all the other collections.

File name: 1.2.11.cpp

```
#include <vector>
#include <iostream>

using namespace std;

class A {
private:
  int number;
public:
  A(int _number) : number(_number) {
    cout << "Normal constructor\n";
  }
  A() {
    cout << "Default constructor\n";
  }
  A(const A& source) {
    number = source.number;
    cout << "Copy constructor\n";
  }
  A & operator = (const A& source) {
    number = source.number;
    cout << "Assignment operator\n";
    return *this;
  }
  ~A() {
    cout << "Destructor\n";
  }
};

int main() {
  vector <A> v1;
  v1.push_back(1);
  cout << "First ready!\n";
  v1.push_back(2);
  cout << "Second ready!\n";
  v1.push_back(3);
  cout << "Third ready!\n";
  return 0;
}
```

1.2.12 deque class

Header: `<deque>`
Definition: `template<class T, class Allocator = std::allocator<T>> class deque;`

The name deque stands for double-ended queue. In many ways, this container is very similar to a vector. **The most important difference is that it does not have to occupy contiguous memory space**. If a `vector` is basically an array inside, **deque is a double-linked list of arrays**. This allows for fast insertion at both ends of the container. There's no need for reallocation, because a new array can always be added on either side. `deque`, similarly to a vector, allows for random access using `operator []`. The `deque` template definition has exactly the same meaning as the `vector` does. The `Allocator` is responsible for the memory model.

1.2.13 deque constructors – size constructors

`deque` has four constructors, virtually identical to those of the `vector`:

```
explicit deque ( const Allocator& = Allocator() );
explicit deque ( size_type n, const T& value= T(), \
  const Allocator& = Allocator() );
emplate <class InputIterator>
deque ( InputIterator first, InputIterator last, \
const Allocator& = Allocator() );
deque ( const deque<T,Allocator>& x );
```

The mode of operation of each of these constructors is identical to their `vector` counterparts. Again we have: the default constructor, the initializing constructor, the iterator constructor and the copy constructor.

On this and the following slides you will see examples of deque's usage of constructors.

File name: 1.2.13.cpp

```
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    deque<int> d1(10, 0);
    cout<<"Size: "<<d1.size()<<endl;
    for(unsigned i = 0; i < d1.size(); ++i)
    {
        cout<< d1[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

1.2.14 deque – iterator constructors

Iterator constructor example:

File name: 1.2.14.cpp

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
  deque <int> d1(10,0);
  cout << "Size: " << d1.size() << endl;
  for (unsigned i = 0; i < d1.size(); ++i) {
    cout << d1[i] << " ";
  }

  cout << endl;
  return 0;
}
```

1.2.15 deque – copy constructors

An example of copy constructor usage for the `deque` class. Elements which might be stored inside a `deque` require the same consideration as those stored inside a `vector`. They need: a proper constructor, a copy constructor and an assignment operator, to work as expected.

File name: 1.2.15.cpp

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
  int a1[] = {1,2,3,4,5,6,7,8,9,10};
  deque <int> d1(a1, a1+10);
  for (unsigned i = 0; i < d1.size(); ++i) {
    cout << d1[i] << " ";
  }
  cout << endl;
  return 0;
}
```

1.2.16 `list`

Header: `<list>`
Definition: `template<class T, class Allocator = std::allocator<T>> class list;`

The `list` container is an implementation of the double-linked list principle. Each element has pointers leading to the next and the previous ones in a `list` sequence. The storage is **not contiguous**; it's likely that each element will be placed in a completely different memory area.

The `list` container allows for fast insertion and deletion anywhere inside the range of its elements. Generally, all operations related to the change in the sequence of elements are less costly than in `vector` and `deque`. **The price to pay for this advantage is the lack of the random access mechanism – operator[]**. But of course, it's still possible to iterate through the collection using a general STL mechanism – the `iterator`. Another drawback is the additional data required to keep linking information for each of the elements. **This results in additional memory consumption**. In specific cases this might be a significant problem.

1.3 Iterators

1.3.1 Iterators

The `iterator` is in many ways similar to the concept of the **pointer**, and in some cases, they can be used interchangeably.

As we stated earlier, there are **five kinds of iterators**. Each category is defined by the ability to perform a chosen set of operations. The table on the right tries to summarize all the information about the different characteristics of `iterators`.

1.3.2 Containers and iterators

Every container is made up of four members (types) related to iterators:

- `iterator` – read/write iterator type;
- `const_iterator` – read-only iterator type;
- `reverse_iterator` – reverse iterator type (iterates from the end to the beginning)
- `const_reverse_iterator` – as above, but read only.

Although the members have the same names throughout the collections, they are different in each container type, and they must facilitate different types of containers.

For example, `vector` and `deque` support random access `iterators`, while a list only supports bidirectional ones. And to top it all off, there is the internal structure of the container to consider, so you can use a `vector` iterator to iterate through a `list`, and vice versa. But on the other hand, when a function or a method expects a bidirectional `iterator`, you can use any one, no matter which collection it belongs to.

Look at the example of different `iterator` declarations. There are three different container objects declared in the code on the right, and for each of them, two `iterators` have been created – normal and const. At the moment, none of these `iterators` are initialized, and therefore cannot be used. Exactly the same is true for uninitialized pointers. We're going to initialize them on the next slide.

File name: 1.3.2.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    vector <int> v(10);
    deque <int> d(10);
    list <int> l(10);

    int i = 1;
```

```
    vector<int>::iterator itV;
    for(itV = v.begin()  ; itV != v.end(); ++itV,++i)
    {
        *itV = i;
    }
    for(itV = v.begin();  itV != v.end(); ++itV)
    {
        cout << *itV << " ";
    }
    cout<<endl;

    deque<int>::iterator itD = d.begin();
    for(itD = d.begin()  ; itD != d.end(); ++itD,++i)
    {
        *itD = i;
    }
    for( itD = d.begin() ; itD != d.end(); ++itD)
    {
        cout << *itD << " ";
    }
    cout<<endl;

    list<int>::iterator itL = l.begin();
    for( ; itL != l.end(); ++itL,++i)
    {
        *itL = i;
    }
    for( itL = l.begin() ; itL != l.end(); ++itL)
    {
        cout << *itL << " ";
    }
    cout<<endl;
    return 0;
}
```

1.3.3 Initialization of iterators

So, we need to initialize iterators in order to use them. To do this, we need to get the iterator value from the container. There are four methods (check out table 1 if you have any queries) that can do that:

```
begin()
end()
rbegin()
rend()
```

These methods are available for all the containers, but the values returned for each of them are different. Each method comes in two variations – normal and 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;
```

Important:

For `vector` and `deque`, these methods return random access iterators, but the list only supports a bidirectional iterator.

- the `begin()` method returns the iterator that points to the first element of the collection;
- the `end()` method returns the iterator that refers to the past-the-end-element. If a container has n elements, this value will be marked n+1 (non-existent). In practice, this just means the end of the collection. But you must remember that the end of a collection does not mean the last element, but the first element after the last;
- the `rbegin()` method means reverse begin – it returns the iterator that points to the last element of the collection;
- the `rend()` method means reverse end – it returns the iterator that refers to the element before the first element of the container – this value indicates the end of the collection in reverse order.

Past-the-end element

The past-the-end element is a virtual element located after the last element of the collection. It indicates the end of the collection.

File name: 1.3.3.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main() {
  vector <int> v(10);
  deque <int> d(10);
  list <int> l(10);
```

```
  int i = 1;

  vector <int> :: iterator itv;
  for (itv = v.begin(); itv != v.end(); ++itv, ++i) {
    *itv = i;
  }

  for (itv = v.begin(); itv != v.end(); ++itv) {
    cout << *itv << " ";
  }
  cout << endl;

  deque <int> :: iterator itd = d.begin();
  for (itd = d.begin(); itd != d.end(); ++itd, ++i) {
    *itd = i;
  }

  for (itd = d.begin(); itd !=d.end(); ++itd) {
    cout << *itd << " ";
  }
  cout << endl;

  list <int> :: iterator itl = l.begin();
  for (itl = l.begin(); itl != l.end(); ++itl, ++i) {
    *itl = i;
  }

  for (itl = l.begin(); itl != l.end(); ++itl) {
    cout << *itl << " ";
  }
  cout << endl;
  return 0;
}
```

1.3.4 Iterators usage examples – reverse iterators

File name: 1.3.4.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;
```

```
int main() {
  vector <int> v(10);
  deque <int> d(10);
  list <int> l(10);

  int i = 1;
  vector <int> :: iterator itv;
  for (itv = v.begin(); itv != v.end(); ++itv, ++i) {
    *itv = i;
  }

  for (vector <int> :: reverse_iterator it = v.rbegin(); \
      it != v.rend(); ++it) {
    cout << *it << " ";
  }
  cout << endl;

  i = 1;
  deque <int> :: iterator itd = d.begin();
  for (itd = d.begin(); itd != d.end(); ++itd, ++i) {
    *itd = i;
  }
  for (deque <int> :: reverse_iterator it = d.rbegin(); \
      it != d.rend(); ++it) {
    cout << *it << " ";
  }
  cout << endl;

  i = 1;
  list <int> :: iterator itl = l.begin();
  for (; itl != l.end(); ++itl, ++i) {
    *itl = i;
  }

  for (list <int> :: reverse_iterator it = l.rbegin(); \
      it != l.rend(); ++it) {
    cout << *it << " ";
  }
  cout << endl;
  return 0;
}
```

1.3.5 Iterators usage examples – const iterators

File name: 1.3.5.cpp

```
#include <iostream>
#include <vector>
#include <deque>
#include <list>

using namespace std;

int main() {
  int a[] {1,2,3,4,5,6,7,8,9,10};
  vector <int> v(a,a+10);
  deque <int> d(a,a+10);
  list <int> l(a,a+10);

  for (deque <int> :: const_iterator it = d.begin(); \
      it != d.end(); ++it) {
    cout << *it << " ";
  }
  cout << endl;

  for (deque <int> :: const_iterator it = d.begin(); \
      it != d.end(); ++it) {
    cout << *it << " ";
  }
  cout << endl;

  for (list <int> :: const_iterator it = l.begin(); \
      it != l.end(); ++it) {
    cout << *it << " ";
  }
  cout << endl;
  return 0;
}
```

1.3.6 Iterators usage examples

Iterators usage examples – `const iterators` – incorrect scenario. The example illustrated in the code cannot be compiled. The compiler doesn't allow anything to be assigned to an element referred to by a `constant iterator`.

File name: 1.3.6.cpp

```
#include <list>
#include <vector>
```

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
  int a[] = {1,2,3,4,5,6,7,8,9,10};

  vector<int> v(a, a+10);
  deque<int> d(a, a+10);
  list<int> l(a, a+10);

  vector<int> :: const_iterator it1 = v.begin();
  *it1 = *it1 + 1;
  deque<int> :: const_iterator it2 = d.begin();
  *it2 = *it2 + 1;
  list<int> :: const_iterator it3 = l.begin();
  *it3 = *it3 + 1;
  return 0;
}
```

1.3.7 Other collections

Iterators of other types of containers work in the same manner. The most important factor in the proper usage of iterators is the iterator type. The most common are:

- **random access**
- **bidirectional**

Remember that you can do more with the first type than the second. Iterators allow you to traverse through collections and manipulate their elements regardless of the collection type. This is a common interface for using STL containers. **Iterators make it relatively easy to switch from one type of container to another without any heavy impact on the source code**. All the STL containers provide iterators, which is why it's so important to fully understand them.

1.4 Operation

1.4.1 size and max_size

Name: `size`
Signature:
`size_type size () const;`
Parameters:
None
Return Value: The number of elements which are currently stored inside a collection.
Description:
This method returns the number of elements which are currently stored inside a container. The size will change each time an element is added to or removed from the container.
Name: `max_size`
Signature:
`size_type max_size () const;`
Parameters:
None
Return Value: The maximum number of elements which can be held inside a container.
Description:
The method returns the maximum physical capacity of a container. This value might depend on the STL library implementation or an operating system, and will always be constant in the same environment.

Example: The example shows the basic usage of the `size()` and `max_size()` methods. As you can see, the size changes after inserting and removing elements from the container. On the other hand, the maximum size remains constant.

File name: 1.4.1.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main() {
  int a[] = {1,2,3,4,5,6,7,8,9,10};

  vector<int> v(a, a+10);
  deque<int> d(a, a+10);
  list<int> l(a,a+10);
  cout << "Size of vector, deque, list: " << v.size() << " " \
    << d.size() << " " << l.size() << " " << endl;
  cout << "Max size of vector, deque, list: " << v.max_size() \
    << " " << d.max_size() << " " << l.max_size() << endl;
```

```
  v.push_back(11);
  d.push_back(11);
  l.push_back(11);
  cout << "Size of vector, deque, list:" << v.size() << " " \
    << d.size() << " " << l.size() << endl;
  cout << "Max size of vector, deque, list: " << v.max_size() \
    << " " << d.max_size() << " " << l.max_size() << endl;

  v.pop_back();
  d.pop_back();
  l.pop_back();
  cout << "Size of vector, deque, list:" << v.size() \
    << " " << d.size() << " " << l.size() << endl;
  cout << "Max size of vector, deque, list: " << v.max_size() \
    << " " << d.max_size() << " " << l.max_size() << endl;

  return 0;
}
```

1.4.2 empty() and resize()

Name: `empty`
Signature:
```
bool empty () const;
```
Parameters:
None
Return Value: The method returns true if a container is empty and false otherwise.
Description:
This method is used to indicate whether a container is empty or not. You should use this method instead of calling `size()` to check if the list container is empty. Using calling `size()` for a list might result in linear time performance for some STL implementations, instead of constant ones.
Name: `resize`
Signature:
```
void resize (size_type sz, T c = T() );
```
Parameters:

`sz`: the new size of a container
`c`: the value to copy in order to add elements into a container when the new size (sz) is greater than the old size
Return Value: None
Description:
This method changes the current size of a container, either by causing it to grow or shrink. The new size is provided by the parameter `sz`.

If the new size is greater than the old size, new elements are added to the container. Those new elements are created by copying parameter c, or, if c is not provided, by copying the default value for a particular type of element. The number of elements added is expressed by a simple formula: new size minus old size.

If the new size is smaller than the old size, the collection will shrink. All elements between the new size and the old size will cease to exist – which might cause their destructors to be called.

Example: In the example, we use the `empty()` method to check whenever containers are eligible for resizing. You must remember that the `resize()` method sets the current size of a container to the exact value provided by its first argument. This argument `sz` is not a delta, but the exact size to be set. Another thing to remember is that `sz` is `unsigned`, so providing a negative value might cause the container not to shrink, but rather to grow close to its limit – `max_size()` .

File name: 1.4.2.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main() {
  vector<int> v;
  deque<int> d;
```

```
  list<int> l;
  cout << "Size of vector, deque, list: " << v.size() << " " \
       << d.size() << " " <<  l.size() << " " << endl;
  if (v.empty()) {
    v.resize(10);
  }
  if (d.empty()) {
    d.resize(10);
  }
  if (l.empty()) {
    l.resize(10);
  }
  cout << "Size of vector, deque, list: " << v.size() << " " \
       << d.size() << " " <<  l.size() << " " << endl;

}
```

1.4.3 vector::capacity() - vector only

Name: `vector<T>::capacity`
Signature:
`size_type capacity () const;`
Parameters:
None
Return Value: The total amount of space for elements currently allocated to a particular vector
Description:

This method is present in the vector class only.

The `vector` class has two methods related to its size. The first one is `size`, which you already know. The second one is capacity.

`capacity` is the total number of slots inside a vector which are currently allocated. Some of the slots might be used; some of them might be free.

`capacity` is always greater than or equal to `size`. As we said earlier, the `vector` occupies a **contiguous memory area**. This approach has some limitations. When a new element is inserted into a `vector`, its size is increased and the whole structure needs to be reallocated. In practice, such an approach isn't very effective. In order to limit the number of reallocations, the vector is equipped with a capacity capability.

Each time a new element is put into the `vector`, it first checks to see if there is enough capacity. If not, the vector is reallocated, but the new capacity is usually larger than the old capacity + 1. So, the next insertion/addition of an element will not require any reallocation of the vector. This approach improves the performance of the vector.

File name: 1.4.3.cpp

```
#include <vector>
#include <iostream>

using namespace std;

int main() {
  vector<int> v;
  cout << "Size and capacity: " << v.size() << " " << \
          v.capacity() << endl;
  for (int i = 0; i < 20; i++) {
    v.push_back(i);
    cout << "Size and capacity: " << v.size() << \
         " " << v.capacity() << endl;
  }
  return 0;
}
```

1.4.4 vector::reserve() vector only

Name: `vector<T>::reserve`
Signature:
```
void reserve (size_type n);
```
Parameters:
`n`: the minimum value of capacity to be requested
Return Value: None
Description:

This method allocates additional space for elements inside a vector. If the newly requested capacity `n` is greater than the current capacity, reallocation is enforced. The new effective capacity will be at least as large as the one requested.

If reallocation happens, all iterators are invalidated. If a requested capacity is lower than the current capacity, the method will do nothing.

Calling this method will never affect the elements already placed in the vector. This method is a way to prepare the vector to accept a certain number of elements. Reserving space beforehand eliminates the need for reallocation after each insertion.

File name: 1.4.4.cpp

```
#include <vector>
#include <iostream>

using namespace std;

int main() {
  vector<int> v;
  cout << "Size and capacity: " << v.size() << " " << \
       v.capacity() << endl;
  v.reserve(15);
  cout << "Size and capacity: " << v.size() << " " << \
       v.capacity() << endl;

  cout << "Adding elements" << endl;
  for(int i = 0; i < 10; i++) {
    v.push_back(i);
    cout << "Size and capacity: " << v.size() << " " << \
         v.capacity() << endl;
  }

  cout << "Trying to shrink ..." << endl;
  v.reserve(10);
  cout << "Size and capacity: " << v.size() << " " << \
       v.capacity() << endl;
  return 0;
}
```

1.4.5 back() and front()

Name: `front`
Signature:
`reference front (); const_reference front () const;`
Parameters:
None
Return Value: A reference to the first element in a container
Description:
this method returns a reference to the first element in a container. The reference might be normal or constant: it all depends on the calling context. The purpose of this method is to retrieve the first value from the container. Because the element is returned by a reference, it's possible to modify it – a call to `front` can become the target of the assignment – the `l-value`. The returned element is not removed from the container.
Name: `back`
Signature:
`reference back (); const_reference back () const;`
Parameters:
None
Return Value: A reference to the last element in a container
Description:
this method returns a reference to the **last element** in a container. The reference might be normal or constant – it all depends on the calling context. The purpose of this method is to retrieve the last value from the container. Because the element is returned by a reference, it's possible to modify it – a call to `back` can become the target of the assignment – the `l-value`. The returned element is not removed from the container.
File name: 1.4.5.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template<class I>
void print (const I& start, const I& end) {
  I it;
  for (it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}

int main() {
  int a[] = {1,2,3,4,5,6,7,8,9,10};
```

```
  vector<int> v(a,a+10);
  deque<int> d(a,a+10);
  list<int> l(a, a+10);

  cout << "Size of vector, deque, list: " << v.size() << " " \
       << d.size() << " " << l.size() << endl;
  cout << "Values at front() (vector, deque, list): " << v.front() \
       << " " << l.front() << endl;
  cout << "Size of vector, deque, list: " << v.size() << " " << d.size() \
       << " " << l.size() << endl;

  v.front() = 100;
  d.front() = 101;
  l.front() = 102;

  print(v.begin(), v.end());
  print(d.begin(), d.end());
  print(l.begin(), l.end());

  cout << endl;
  v.back() = 200;
  d.back() = 201;
  l.back() = 202;

  print(v.begin(), v.end());
  print(d.begin(), d.end());
  print(l.begin(), l.end());

  return 0;
}
```

1.4.6 operator[] and at() `vector and deque only`

Name: `operator[]`
Signature:
`reference operator[] (size_type n); const_reference operator[] ( size_type n) const`
Parameters:
`n`: the index of the element to access
Return Value: A reference to the element of index `n`
Description:

`operator []` allows containers (`vector` and `deque`0 in this case) to be treated in a similar way to `arrays`. Each element of these collections can be accessed using [] – square brackets. The correct range of indexes is 0 to size -1, just as in the case of an ordinary array.

Because `operator []` returns a reference, the accessed element can be used as the l-value and the r-value.

`operator []` **does not check** if index n is in the proper range – 0 to size – 1, so it's possible to access an element which is not in fact in the container. This may lead to unpredictable results.

In comparison, the method `at()` performs exactly the same task, but with **index range checking**. When used as the l-value, `operator[]` can only change an already stored element. It's not possible to add an element to a collection using this operator – it doesn't change the size of the container.

Name: `at`
Signature:
`reference at ( size_type n );\ const_reference at ( size_type n ) const`
Parameters:
`n`: the index of the element to be accessed
Return Value: A reference to the element of index `n`
Description:

the method `at()` is used to retrieve an element from the STL container (`vector` and `deque`). It retrieves the value stored under the index `n`. This method returns a reference, which means it can be used as the `l-value` as well as the `r-value`.

`at()` is very similar in its behavior to `operator[]` . The only difference is that `at()` performs a **range check** on parameter `n`, and if `n` is out of range, an `out_of_range exception` is thrown. When used as an `l-value`, `at()` can only change an already stored element. It's not possible to add an element to a collection using this method – it doesn't change the size of the container.

File name: 1.4.6.cpp

```
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template <class C>
void print1 (const C& container) {
  for (unsigned i = 0; i < container.size(); i++) {
    cout << container[i] << " ";
  }
```

```
  cout << endl;
}

template <class C>
void print2 (const C& container) {
  for (unsigned i = 0; i < container.size(); i++) {
    cout << container.at(i) << " ";
  }
  cout << endl;
}

int main() {
  int a[] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(10);
  deque<int> d(10);

  for (unsigned i = 0; i < 10; ++i) {
    d[i] = a[i];
    v[i] = a[i];
  }
  cout << "Accessing by operator []" << endl;
  print1 (v);
  print1 (d);
  cout << "Accessing out of range element: \n";
  cout << v[10] << " " << d[10] << endl << endl;

  cout << "Accessing by operator at()" << endl;
  print2 (v);
  print2 (d);
  cout << "Accessing out of range element: \n";
  try {
    cout << v.at(10) << endl;
  }
  catch (out_of_range & ex) {
    cout << ex.what() << endl;
  }

  try {
    cout << d.at(10) << endl;
  }
  catch (out_of_range & ex) {
    cout << ex.what() << endl;
  }
  return 0;
```

```
}
```

1.4.7 assign()

Name: `assign`
Signature:
`template <class InputIterator> void assign ( InputIterator first, InputIterator last ); void assign`
Parameters:

`first, last`: the input iterators which provide a collection of input elements. The `assign` method will copy all the elements from this range, including `first` and excluding `last`. Because `first` and `last` are of the `InputIterator` type, virtually any type of iterator can be used in the call

`n`: the number of times the value will be copied to fill the container.

`u`: the value to be copied

Return Value: None
Description:

This method assigns new values to an already existing container. The whole old content of the container is dropped and deleted.

The new content is provided by one of two means: by either the iterator's range or value, which will fill a certain number of elements.

In both cases, the new elements which are stored inside the collections are obtained by copying the source values. As a result of the dropping of all the old elements, it's possible to use a source container of a different size to the target one. After the assignment, they will both have the same size.

File name: 1.4.7.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template <class I>
void print(const I& start, const I& end) {
  I it;
  for (it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}

int main() {
  int a[] = {1,2,3,4,5,6,7,8,9,10};

  vector <int> v(a, a+5);
  deque <int> d(a,a+5);
  list <int> l(a, a+5);
```

```
  print(v.begin(), v.end());
  print(d.begin(), d.end());
  print(l.begin(), l.end());

  cout << "Assigning a new content: \n";
  v.assign(a, a+10);
  d.assign(a, a+10);
  l.assign(a, a+10);

  print(v.begin(), v.end());
  print(d.begin(), d.end());
  print(l.begin(), l.end());

  cout << "Assigning a new content II:\n";
  v.assign(3, 100);
  d.assign(3,1000);
  l.assign(3,10000);

  print(v.begin(), v.end());
  print(d.begin(), d.end());
  print(l.begin(), l.end());

  return 0;
}
```

1.4.8 insert()

Name: `insert`
Signature:
`iterator insert ( iterator position, const T& x ); void insert ( iterator position, size_type n, co`
Parameters:

`position` – the `position` in the container at which the insertion of an element (or elements) is to be performed. For `deque` and `vector`, this is `RandomAccessIterator`, while in the case of a `list`, `BidirectionalIterator` is used;

`x` – the value to be inserted;

`n` – the number of `x` values to be inserted;

`first, last` – the `iterators` specifying the range of elements to be inserted into the container. As usual, the range includes `first` and excludes `last`.

Return Value: The first version of this method returns an `iterator` to a newly inserted object if the insertion is successful. Other versions do not return anything.
Description:

The method `insert()` performs an insertion into a container. There are three variants of this method, as stated in the signature section. Inserting an element into a container will cause the container to grow. This leads to different consequences, depending on the type of collection:

- `vector` – when an increase in size causes it to reallocate (not enough capacity left) all iterators, references and pointers will be invalidated;
- `deque` – all iterators will be invalidated, references also, unless insertion at the beginning or end takes place;
- `list` – the iterators and references remain.

File name: 1.4.8.cpp

```
#include <vector>
#include <deque>
#include <list>
#include <iostream>

using namespace std;

template <class I>
void print(const I& start, const I& end) {
  I it;
  for (it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}

int main() {
```

```
  int a[] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(a, a+10);
  list<int> l(a, a+10);
  deque<int> d(a, a+10);

  vector<int>::iterator it = v.insert(v.begin()+5, 100);
  list<int>::iterator it1 = l.insert(l.begin(), 100);
  deque<int>::iterator it2 = d.insert(d.begin(), 100);


  print(v.begin(), v.end());
  cout << "Inserted element: " << *it << endl;
  cout << "Size: " << v.size() << endl;
  vector<int> v2;
  v2.insert(v2.begin(), v.rbegin(), v.rend());
  print(v2.begin(), v2.end());

  vector<int> v3(v.begin(), v.begin() + 5);
  v3.insert(v3.end(), 3, 100);
  print(v3.begin(), v3.end());

  list<int> l1;
  l1.insert(l1.begin(), l.begin(), l.end());
  return 0;
}
```

1.4.9 erase()

Name: `erase`
Signature:
`iterator erase ( iterator position ); iterator erase ( iterator first, iterator last );`
Parameters:

`position` – the iterator pointing to the element to be erased

`first, last` – the iterators which specify the range of elements to be erased. As usual, the range includes first and excludes last.

Return Value: An iterator to the first element after the last removed element, or `end` if the operation removes the last element in the collection.
Description:

This function removes an element or a range of elements from a collection. It's important to note that it uses iterators, and not the index value, nor the value of the element itself. After the operation, the size of the collection is decreased. During the removal of the elements, the destructors are called.

File name: 1.4.9.cpp

```
#include <vector>
#include <list>
#include <deque>
#include <iostream>

using namespace std;

template <class I>
void print (const I& start, const I& end) {
  I it;
  for (it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}
int main() {
  int a[] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(a, a+10);
  list<int> l(a, a+10);
  deque<int> d(a, a+10);

  print(v.begin(), v.end());
  print(l.begin(), l.end());
  print(d.begin(), d.end());

  cout << "Ereasing elements: \n";
  v.erase(v.begin() + 3);
  d.erase(d.begin() + 3);
```

```
  list<int>::iterator it = l.begin();
  ++it; ++it; ++it;
  it = l.erase(it);

  print(v.begin(), v.end());
  print(l.begin(), l.end());
  print(d.begin(), d.end());

  cout << "Ereasing elements: \n";
  v.erase(v.begin()+3, v.end());
  d.erase(d.begin()+3, d.end());
  l.erase(it, l.end());

  print(v.begin(), v.end());
  print(l.begin(), l.end());
  print(d.begin(), d.end());

  return 0;
}
```

1.4.10 swap()

Name: `swap`
Signature:
`void vector::swap ( vector<T,Allocator>& vec ); void deque::swap ( deque<T,Allocator>& dqe ); void`
Parameters:
`vec, deq, lst`: another collection of the same type as this one.
Return Value: None
Description:
This method swaps the entire content between two collections of the same type (`list <-> list, vector <-> vector, deque <-> deque`). Although the types of collections must be **the same**, their **sizes may differ**. After the change, all the existing iterators are still valid, but they of course point to different containers. After the swap operations, the calling container will be made up of elements from the source collection, and vice versa.
File name: 1.4.10.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template <class I>
void print(const I& start, const I& end) {
  I it;
  for (it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}

int main() {
  int a[] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v1(a,a+5);
  list<int> l1(a,a+5);
  deque<int> d1(a,a+5);

  vector<int> v2(a+5, a+10);
  list<int> l2(a+5,a+10);
  deque<int> d2(a+5,a+10);

  v1.swap(v2);
  l1.swap(l2);
  d1.swap(d2);
```

```
  return 0;

}
```

1.4.11 clear()

Name: `clear`
Signature:
`void clear ();`
Parameters:
None
Return Value: None
Description:
This function removes all the elements from the collection and sets its size to 0. During the removal, the destructors are called.
File name: 1.4.11.cpp

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    vector <int> v(a,a+10);
    deque <int> d(a,a+10);
    list <int> l(a,a+10);

    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());

    cout<<"Clearing collections:\n";
    v.clear();
    d.clear();
    l.clear();
```

```
    v.push_back(100);
    d.push_back(100);
    l.push_back(100);
    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());

    return 0;
}
```

1.4.12 push_back() and pop_back()

Name: `pop_back()`
Signature:
```
void push_back (const T& x);
```
Parameters:
`x`: the value which will be used to create a new element (by copying) inside a container.
Return Value: None
Description:
The function `push_back()` adds a new value to a container. The value is added at the end (the back) of the container, and increases the size of the container by one. Different types of containers react differently:

- if a `vector` has enough capacity, the item is just added to it, no reallocation is performed, and all obtained iterators remain valid
- if there is not enough capacity left, a reallocation is performed, which invalidates all iterators
- In the case of `deque`, all iterators are invalidated
- For a `list` container, all iterators are left unaffected.

Name: `pop_back`
Signature:
```
void pop_back();
```
Parameters:
None
Return Value: None
Description:
This function removes an element from the tail of the container. Basically, it is the opposite method to `push_back()`. During element removal, its destructor is called, and the container size is reduced by one. In the case of a vector call, this method invalidates all iterators, pointers and references referring to the removed element.

It's also worth noticing that `pop_back()` only removes the value, and the value is not returned. This method cannot be used as the `l-value`. To obtain a value, you should use `back() first`.

File name: 1.4.12.cpp

```
#include <vector>
#include <deque>
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
```

```
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    vector <int> v;
    deque <int> d;
    list <int> l;

    for(unsigned i = 0; i < 10; ++i)
    {
        v.push_back(i);
        d.push_back(i);
        l.push_back(i);
    }
    cout<<"Vector: ";    print(v.begin(), v.end());
    cout<<"Deque:  ";    print(d.begin(), d.end());
    cout<<"List:   ";    print(l.begin(), l.end());

    for(unsigned i = 0; i < 5; ++i)
    {
        v.pop_back();
        d.pop_back();
        l.pop_back();
    }

    cout<<"Vector: ";    print(v.begin(), v.end());
    cout<<"Deque:  ";    print(d.begin(), d.end());
    cout<<"List:   ";    print(l.begin(), l.end());

    return 0;
}
```

1.4.13 push_front() and pop_front() `list` and `deque` only

Name: `push_front`
Signature:
```
void push_front(const T& x);
```
Parameters:
`x` – the value which will be used to create a new element (by copying) inside a container.
Return Value: None
Description:
The function `push_front()` adds a new value to a container. The value is added at the beginning (the front) of the container, and increases the size of the container by one. Different types of containers react differently:

- in the case of `deque`, all iterators are invalidated
- for a `list` container, all iterators are left unaffected.

Name: `pop_front`
Signature:
```
void pop_front();
```
Parameters:
None
Return Value: None
Description:
This function removes an element from the beginning of the container. Basically, it's the opposite method to `push_front()`. During element removal, its destructor is called, and the container size is reduced by one. It's also worth noticing that `pop_front()` only removes the value, and the value is not returned. This method cannot be used as the l-value. To obtain a value, you should use `front()` first.

File name: 1.4.13.cpp

```
#include <deque>
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
```

```
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    deque <int> d;
    list <int> l;

    for(unsigned i = 0; i < 10; ++i)
    {
        d.push_front(i);
        l.push_front(i);
    }
    cout<<"Deque:  ";    print(d.begin(), d.end());
    cout<<"List:   ";    print(l.begin(), l.end());

    for(unsigned i = 0; i < 5; ++i)
    {
        d.pop_front();
        l.pop_front();
    }
    cout<<"Deque:  ";    print(d.begin(), d.end());
    cout<<"List:   ";    print(l.begin(), l.end());

    return 0;
}
```

1.4.14 splice() – list only

Name: `splice`
Signature:
`void splice ( iterator position, list<T,Allocator>& x ); void splice ( iterator position, list<T,Al`
Parameters:
`position` – the position in the calling list where the elements will be inserted;
`x` – the list from which the elements will be moved to the calling list
`i` – the iterator to a single element from the source list, which will be moved to the calling list
`first, last` – the iterators which define the range of elements to be moved from the source list to the destination. The range includes `first` and excludes `last`.
Return Value: None
Description:
This method moves elements from a list specified as parameter `x`, and inserts them into the list container which calls the method. The target list size increases by the number of elements moved, while the source list size decreases accordingly. There are three versions of this method. A method which moves the whole content of the source container. A method which moves one, and only one, element specified by the iterator.A method which moves a range of elements specified by the iterators. In all cases, there's no object destruction or construction involved during the call.
File name: 1.4.14.cpp

```
#include <list>
#include <iostream>

using namespace std;

template <class I>
void print (const I& start, const I& end) {
  for (I it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}

int main() {
  int a[] = {1,2,3,4,5};
  int b[] = {6,7,8,9,10};
  int c[] = {11,12,13,14,15};
  list<int> l1(a,a+5);
  list<int> l2(b,b+5);
  list<int> l3(c,c+5);

  l2.splice(l2.end(), l3);
  print(l2.begin(), l2.end());
  cout << "Size of source list l3: " << l3.size() << endl;
```

```
  list<int> :: iterator it = l2.begin();
  advance(it,9);
  l1.splice(l1.end(), l2, it);
  print(l1.begin(), l1.end());
  cout << "Size of source list l2: " << l2.size() << endl;

  it = l1.end();
  advance(it,-1);
  l1.splice(it, l2, l2.begin(), l2.end());
  print(l1.begin(), l1.end());
  cout << "Size of source list l2: " << l2.size() << endl;

  return 0;
}
```

1.4.15 remove() and remove_if() – list only

Name: `remove(list)`
Signature:
`void remove (const T& value);`
Parameters:
`value` – the value of the element to be removed from the list. It's the same type as that used during the list declaration.
Return Value: None
Description:
This function removes from the list all the elements equal to the values provided as the parameters. During the removal, the destructors are called. This function works in a different way in comparison to `erase` which uses iterators.
Name: `remove_if(list)`
Signature:
`template <class Predicate>\ void remove_if ( Predicate pred );`
Parameters:
`pred` – a unary predicate (one argument function, or function object) which takes an argument of the same type as the elements of the list. The predicate should return `true` for elements which are to be removed, and `false` for all others.
Return Value: None
Description:
The function `remove_if()` performs a conditional object deletion. The method calls the provided predicate for every element stored inside the list. If the predicate returns `true`, the element is eligible for removal. During the removal, the destructors are called and the size of the list decreases.
File name: 1.4.15.cpp

```
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print(const I& start, const I& end) {
  for (I it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout <<endl;
}

struct DeleteOdd {
  bool operator() (int value) {
    if (value % 2 > 0)
      return true;
    return false;
```

```
  }
};

bool deleteEven(int value) {
  if (value % 2 == 0) {
    return true;
  }
  return false;
}

int main() {
  int a[] = {1,2,1,3,2,3,4,3,4,7,8,9,6,6,5,8,9,10};

  list<int> l1(a, a+18);
  list<int> l2(a, a+18);
  print(l1.begin(), l1.end());

  l1.remove_if(DeleteOdd());
  cout << "All odd numbers have been deleted" << endl;
  print(l1.begin(), l1.end());

  l2.remove_if(deleteEven);
  cout << "All even numbers have been deleted" << endl;
  print(l2.begin(), l2.end());

  return 0;

}
```

1.4.16 unique() – list only

Name: `unique(list)`
Signature:
`void unique ( ); template <class BinaryPredicate> void unique ( BinaryPredicate binary_pred );`
Parameters:

`binary_pred` – binary predicate – a two-argument function or proper functional object, which performs a comparison between two elements of the list in order to decide whether they are alike or not. If they are found to be alike, the second element is removed.

Return Value: None
Description:

The function `unique()` removes consecutive duplicates from the list. Basically, the function makes a comparison between every two directly subsequent elements (starting from the beginning of the list) and if they're found to be alike, the second element is deleted. The second version of this function does exactly the same thing, but it uses a binary predicate to decide whether the objects are equal or not.

For every element eligible for deletion, its constructor is called, and the size of the list is reduced. To supply the predicate to the method, you must use either the function pointer, or the functional object instance.

File name: 1.4.16.cpp

```
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print(const I& start, const I& end) {
  for (I it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}

bool compareInt(double v1, double v2) {
  if ((int)v1 == (int)v2) {
    return true;
  }
  return false;
}

int main() {
  int a[] = {1,2,1,3,2,3,4,3,4,7,8,9,6,6,5,8,9,10};

  list<int> l1(a,a+18);
  list<int> l2(a,a+18);
  print(l1.begin(), l1.end());
```

```
  cout << "Deleting all subsequent duplicates:" << endl;
  l1.unique();
  print(l1.begin(), l1.end());

  cout << "Deleting all subsequent duplicates from sorted list:" << endl;
  l2.sort();
  l2.unique();
  print(l2.begin(), l2.end());

  cout << "Test function with double type";
  double b[] = {1.6,2.3,1.4,3.1,2.4,3.5,4.9,3.2,4.7,7.8,8,9.1,6.2,6.8,\
    5.5,8.4,9.2,10};
  list<double> l3(b,b+18);
  list<double> l4(b,b+18);

  print(l3.begin(), l3.end());
  cout << "Deleting all subsequent duplicates - int comparison" << endl;
  l1.unique(compareInt);
  print(l3.begin(), l3.end());

  cout << "Deleting all subsequent dublicates - int comparison - sorted" << endl;
  l4.sort();
  l4.unique(compareInt);
  print(l4.begin(), l4.end());

  return 0;
}
```

1.4.17 merge() – list only

Name: `merge (list)`
Signature:
`void merge ( list<T,Allocator>& x ); template <class Compare> void merge ( list<T,Allocator>& x, Co`
Parameters:

`x` – the source list, whose elements are to be merged into the list calling the function;
`comp` – the binary predicate used to compare the elements from the source and target lists in order to ensure the proper sequence of elements in the target list.

Return Value: None
Description:

This method performs a merge of two sorted list. In order to do so, two iterators are used: one in the target (calling) list, and the second in the source list. The `merge()` method compares the objects' pointers with those iterators, and if the source object is less than the target, it's removed from the source and placed in the target list at the target iterator position.

If the source object is greater, the insertion iterator advances and the procedure repeats. This operation is repeated until the insertion iterator reaches the `end()` of the target list. At that moment, if there are any elements left in the source list, they're all moved to the target list, and placed at the end.

The second version of the method uses an external comparator in the form of a binary predicate to perform a comparison between the elements from the target and the source. The predicate takes as its first argument an object from the target list, and as the second argument an element from the source list. It should return true if the target is less than the source, and false otherwise. It should perform the strictly weak strict ordering.

During `merge()`, the objects are moved from the source to the target. Effectively, no object is created, copied, or deleted. This function requires both lists to be sorted before it can be called.

File name: 1.4.17.cpp

```
#include <list>
#include <iostream>

using namespace std;
template<class I>
void print(const I& start, const I& end) {
  for(I it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}

bool compare(int a, int b) {
  if (a > b)
    return true;
  return false;
}

int main() {
```

```
  int a[] = {11,2,3,4,5};
  int b[] = {6,7,8,9,10};

  list<int> l1(a, a+5);
  list<int> l2(b, b+5);

  list<int> l3(l1.rbegin(), l1.rend());
  list<int> l4(l2.rbegin(), l2.rend());

  l2.merge(l1);
  print(l2.begin(), l2.end());
  cout << "Size of the source list l1: " << l1.size() << endl;
  l3.merge(l4, compare);
  print(l3.begin(), l3.end());
  cout << "Size of source list l4: " << l4.size() << endl;
  return 0;
}
```

1.4.18 sort() – list only

Name: `sort(list)`
Signature:
`void sort ( ); template <class Compare> void sort ( Compare comp );`
Parameters:

`comp` – the binary predicate used to compare pairs of elements in order to ensure a proper sort order. It takes arguments of the same type as the elements of the list.

Return Value: None
Description:

This method performs the sorting of elements in a lexicographic order – from lowest to highest.

The first version uses the operator < to compare pairs of elements. This operator is available for built-in types. In order to use this function with other types, you must either create the operator for those types, or create a binary predicate which will perform basically the same role. In that case, the second variant of `sort()` must be used. This predicate must perform strict weak ordering. During the sort, there's no deletion, creation, or copying of elements within the list. The objects are moved only.

File name: 1.4.18.cpp

```
#include <list>
#include <iostream>

using namespace std;

template<class I>

void print(const I& start, const I& end) {
  for (I it = start; it != end; ++it) {
    cout << *it << " ";
  }
  cout << endl;
}

bool compare(int v1, int v2) {
  if (v1 > v2) {
    return true;
  }
  return false;
}

int main() {
  int a[] = {1,2,1,3,2,3,4,7,8,9,6,5,8,9,10};
  list<int> l(a,a+15);
  print(l.begin(), l.end());
  cout << "Sorting - ascending" << endl;
  l.sort();
```

```
  print(l.begin(), l.end());
  cout << "Sorting - descending" << endl;
  l.sort(compare);
  print(l.begin(), l.end());
  return 0;
}
```

1.4.19 reverse() – list only

Name: `reverse(list)`
Signature:
`void reverse ( )`
Parameters:
None
Return Value: None
Description:
This method reverses the order of the elements in the `list` container.
File name: 1.4.19.cpp

```
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[]={1,2,1,3,2,3,4,7,8,9,6,5,8,9,10};

    list <int> l1(a,a+15);

    print(l1.begin(), l1.end());

    cout<<"Reversing order"<<endl;
    l1.reverse();
    print(l1.begin(), l1.end());

    return 0;
}
```

1.5 Container adaptors

1.5.1 Container adaptors

In STL terminology, a container adaptor is a class which uses an STL container in order to provide **other functionality**. In our case, these adaptors use sequential collections to **create additional data structures**, like **stacks** or **queues**. The STL provides three types of container adaptors:

- stack;
- queue;
- priority queue.

In the table, you can find all the methods provided by container adaptors. As you can see, their subset is rather **limited in comparison to standard containers**, but their names should mostly be familiar to you. Most of these methods work as a simple proxy, and call the methods of the underlying containers.

In order for adaptors to work, you need to **provide** them with a **proper object container**. This can be done during their initialization – **the adaptor constructor** allows for that. You can choose which container will be used, or rely on a default one, which is different for every adaptor class. Each adaptor class requires a slightly different interface and functionality from its underlying collection.

Generally, **only sequential containers can be used**, but there are other limits related to specific adaptors; we'll describe those limits a little later. You don't have to use STL classes as underlying container types, though. As long as a container provides the interface required by the container adaptor to work, it can be any class.

1.5.2 stack

Name: `<stack>`
Signature:

```
template < class T, class Container = deque<T> > class stack;
```

Parameters:

`T`: the type of the elements stored in the stack
`Container`: the type of the underlying storage container
Return Value: None
Description:

None

The stack class is just an STL implementation of a stack data structure – the LIFO (last in first out) concept. In such a container, you can only add and remove elements, at its one end, which is usually called the top. The top always points to the last inserted element, and only this element can be removed from the stack.

A stack requires the following interface from its internal container:

- `back()`
- `push_back()`
- `pop_back()`

Therefore **all sequential containers (vector, deque, list)** can be used as underlying storage. If an internal container is not specified, a `deque` is used.

1.5.3 Stack object initialization

Name: `constructor`
Signature:

```
explicit stack( const Container& cont = Container() );
```

Parameters:

`cont`: a container provided to serve as internal storage. Its type must be the same as the type defined in the stack template.
`other`: an already existing stack which is used as a template to instantiate the current object. Its internal container must be of the same type as the object being created.
Return Value: None
Description:

First, the constructor creates a stack using the internal container provided. The container does not have to be empty; there might be objects already inside. However, `cont` needs to be of the same type as specified in the stack object declaration – the Container template parameter.

The second constructor is just a copy constructor. Again, the source stack and the stack being created must use internal storage of the same type.

The example shows the correct way to create stack objects.

File name: 1.5.2.cpp

```
#include <iostream>
#include <stack>
#include <deque>
#include <list>
#include <vector>
using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};

    stack <int> s1;

    stack <int> s2(s1);

    deque <int> d1(a1, a1+10);
    stack <int> s3(d1);

    stack<int, list <int> > s4;
    stack<int, vector <int> > s5;

    return 0;
}
```

1.5.4 Stack initialization – the wrong way

1. Not allowed - iterator constructor
2. Not allowed - copy constructor source and target stack object using different storage containers
3. Not allowed - initialization using predefined container – using different storage object than declared

File name: 1.5.3.cpp

```
#include <iostream>
#include <stack>
#include <deque>
#include <list>
#include <vector>
using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};

    stack<int> s1(a1, a1+10);

    stack<int, vector <int> > s2(s1);

    deque <int> d1(a1, a1+10);
    stack<int, vector <int> > s3(d1);

    return 0;
}
```

1.5.5 Stack – assignment operator

Name: `assignment operator =`

Signature:

```
stack<T, Container>& operator=( const stack<T,Container>& other );
```

Parameters:

`other:` the source stack object, its contents are copied and stored in the target stack. Source and target objects must use the same type of internal container.

Return Value: A reference to a calling object (this)

Description:

The assignment operator copies elements from other and places them in the calling object (l-value). As in the case of the copy constructor, the source and the target stack objects must both be created using **the same type of internal storage**.

File name: 1.5.4.cpp

```
#include<iostream>
#include<stack>
#include<deque>
```

```
#include<list>
#include<vector>

using namespace std;

int main()

{
    int a1[]={1,2,3,4,5,6,7,8,9,10};

    deque<int> d1(a1, a1+10);

    stack<int> s1;

    stack<int> s2(d1);

    s1 = s2;

    cout<<s1.size()<<":"<<s2.size()<<endl;

    stack<int, vector<int> > s3;

    return 0;

}
```

1.5.6 Stack – destructor, empty() and size()

Name: `destructor`
Signature:
`~stack();`
Parameters:
None
Return Value: None
Description:
Destroys a `stack` object. The **destructor** of a stack calls the destructors (if applicable) of all objects stored inside the stack, and **deallocates** all used storage.
Name: `empty`
Signature:
`bool empty ( ) const;`
Parameters:
None
Return Value: `true` if the stack size is 0, and `false` otherwise.
Description:

This method tests if the stack size is 0, which means the stack is empty. The function is just a proxy which calls the method of the same name in the underlying container.

Name: `size`

Signature:

```
size_type size ( ) const;
```

Parameters:

None

Return Value: The number of elements currently stored inside the stack.

Description:

Returns the number of elements stored in a stack. The function is just a proxy which calls the method of the same name in the underlying container.

File name: 1.5.5.cpp

```
#include <stack>
#include <iostream>

using namespace std;

int main()
{
    stack <int> s1;
    if (s1.empty())
    {
        s1.push(10);
    }
    cout<<"Size of stack: " << s1.size() << endl;
    return 0;
}
```

1.5.7 Stack – top()

Name: `top`

Signature:

```
value_type& top ( ); \ const value_type& top ( ) const;
```

Parameters:

None

Return Value: A reference to the element at the top of the stack.

Description:

This function returns a reference to the top-most element of the stack. Since the stack is a **LIFO** container, it also means it returns a reference to the last pushed element. `top()` is a proxy method, which means it calls the `back()` method of an underlying container. The member type `value_type` is defined as the type of the elements from the underlying container.

Name: `push`

Signature:

```
void push ( const T& x ); \ const value_type& top ( ) const;
```

Parameters:

`x` – the value to be `pushed` (copied) onto the top of the stack;

`T` – this is the first template parameter – the type of the elements stored inside the container.

Return Value: A reference to the element at the top of the stack.

Description:

The `push()` method adds a new value onto the top of the stack. The new value lies on top of the previously pushed value. This effectively increases the size of the stack by one. The new element is created by coping parameter `x`. This method is a proxy, which means it calls the `push_back()` method in the underlying storage container.

Name: `pop`

Signature:

```
void pop ( );
```

Parameters:

None

Return Value: None

Description:

The `pop()` method removes the top-most element from the stack, and therefore reduces its size by one. The element placed under the current top-most element becomes the new top of the stack.

It's important to remember that the `pop()` call does not return the removed element. If you want to store the value of the top of the stack, call the `top()` method. This method is a proxy, which means it calls the `pop_back()` method in the underlying storage container.

File name: 1.5.6.cpp

```
#include <stack>
#include <iostream>

using namespace std;

int main()
{
    stack <int> s1;

    if (s1.empty())
    {
        cout<< "The stack is empty!\n";
    }

    s1.push(3);
    s1.push(2);
    s1.push(1);

    cout<< "Size: " << s1.size() << endl;
    cout<< "Top element: " << s1.top() << endl;
    cout<< "Size: " << s1.size() << endl;
    s1.pop();
```

```
    cout<< "Top element: " << s1.top() << endl;
    cout<< "Size: " << s1.size() << endl;
    return 0;
}
```

1.5.8 Queue class

Name: <queue>
Signature:
`template < class T, class Container = deque<T> > class queue;`
Parameters:
`T` – the type of the elements stored in the queue;
`Container` – the type of underlying storage container.
Return Value: None
Description:

`queue` is a container which provides FIFO (first in first out) functionality. In the STL, it's implemented as a container adaptor, therefore it requires some other container to work as storage space. As for the FIFO concept, the element can be added (pushed) to one end of the queue (the back) and removed (popped) from the other (the front).

In order for the queue to work, it requires a storage object with the following interface:

`front()`
`back()`
`push_back()`
`pop_front()`

Therefore, the STL containers `deque` and `list` can be used. Of course, any collection which makes use of these methods will be appropriate. If the type of the underlying container is not specified explicitly, `deque` is used (check the class queue signature).

1.5.9 Queue – initialization

Most of the list methods are just proxy methods, calling the method in the underlying container. **Name:** `constructor`
Signature:
`explicit queue( const Container& cont = Container() );\ queue( const queue& other );`
Parameters:

`cont` – the container provided to serve as internal storage. Its type must be the same as the type declared in the queue template;
`other` – the already existing queue, which is used as a template to instantiate the current object; its internal container must be of the same type as the object being created.
Return Value: None
Description:

First, the constructor creates the `queue` using the container provided. The container does not have to be empty; there might be objects already inside. An important fact is that this container must be of the same type

as the internal type specified in the queue object declaration – the Container template parameter.
Second, the constructor is a copy constructor. Again, the source queue object and the queue being created must use the same type of internal storage.

Name: destructor

Signature:

```
~queue();
```

Parameters:

None

Return Value: None

Description:

Destroys the queue object. The queue destructor calls the destructors (if applicable) of all objects stored inside the queue and deallocates all used storage.

Here is the correct way to create queue objects.

File name: 1.5.8.cpp

```
#include <iostream>
#include <queue>
#include <deque>
#include <list>
#include <vector>

using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};
    queue <int> s1;
    queue <int> s2(s1);

    deque <int> d1(a1, a1+10);
    queue <int> s3(d1);

    queue<int, list <int> > s4;
    queue<int, vector <int> > s5;
    return 0;
}
```

1.5.10 Queue initialization – the wrong way

1. Not allowed - iterator constructor
2. Not allowed - copy constructor source and target stack object using different storage containers
3. Not allowed - initialization using predefined container - using different storage object than declared

File name: 1.5.9.cpp

```
#include <queue>
```

```
#include <deque>
#include <list>
#include <vector>

using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};

    queue <int> s1(a1, a1+10);

    queue<int, vector <int> > s2(s1);

    deque <int> d1(a1, a1+10);
    queue<int, vector <int> > s3(d1);

    return 0;
}
```

1.5.11 Queue – assignment operator

Name: `assignment operator =`
Signature:
`queue<T, Container>& operator=( const queue<T,Container>& other );`
Parameters:
`other` – the source queue object, whose contents are copied and stored in a target queue. The source and target objects must use the same type of internal container.
Return Value: A reference to a calling object (*this).
Description:
The assignment operator copies the elements from other and places them in a calling object (l-value). As in the case of the copy constructor, the source and target objects must be created using the same type of internal storage.
File name: 1.5.10.cpp

```
#include <iostream>
#include <queue>
#include <deque>
#include <list>
#include <vector>

using namespace std;

int main()
{
```

```
    int a1[]={1,2,3,4,5,6,7,8,9,10};
    deque <int> d1(a1, a1+10);
    queue <int>s1;
    queue <int> s2(d1);

    s1 = s2;
    cout<<s1.size()<<":"<<s2.size()<<endl;

    queue<int, vector <int> > s3;

    return 0;
}
```

1.5.12 Queue – empty() and size() methods

Name: `empty`
Signature:
`bool empty ( ) const;`
Parameters:
None
Return Value: true if the `queue` size is 0, and false otherwise.
Description:
This method tests if the `queue` size is 0, which means that the queue is empty. The function is just a proxy which calls the method of the same name in the underlying container.
Name: `size`
Signature:
`size_type size ( ) const;`
Parameters:
None
Return Value: The number of elements currently stored inside a `queue`.
Description:
Returns the number of elements stored in the `queue` internal storage. The function is just a proxy which calls the method of the same name in the underlying container.
File name: 1.5.11.cpp

```
#include <queue>
#include <iostream>

using namespace std;

int main()
{
    queue <int> q1;
    if (q1.empty())
    {
```

```
        q1.push(100);
    }
    cout<<"Size of queue: " << q1.size() << endl;
    return 0;
}
```

1.5.13 Queue – front(), back(), push() and pop()

Name: `front`
Signature:
`value_type& front ( );\ const value_type& front ( ) const;`
Parameters:
None
Return Value: A reference to the element at the front of a `queue`.
Description:
This function returns a reference to the front-most element of a queue. This element is the one which has been in the `queue` the longest, and only this element can be removed (popped) from the `queue`. `front()` is a proxy method, which means it calls the `front()` method of the underlying container. The member type `value_type` is defined as the type of the elements from the underlying container.
Name: `back`
Signature:
`value_type& back ( );\ const value_type& back ( ) const;`
Parameters:
None
Return Value: A reference to the last element in the `queue`
Description:
This function returns a reference to the last element of the `queue` – in other words, to the last inserted (pushed) element.
`back()` is a proxy method, which means it calls the method of the same name in the underlying container. The member type `value_type` is defined as the type of the elements from the underlying container – the `T` parameter of the `queue` template. The last element of the `queue` cannot be removed.
Name: `push`
Signature:
`void push ( const T& x );`
Parameters:
`x` – the value to be inserted (pushed) at the end of the queue;
`T` – this is the first template parameter – the type of the elements stored inside the container.
Return Value: None
Description:
The `push()` method adds a new element to the `queue`. The new value is placed after the previously pushed one, and becomes the last element of the `queue`. This effectively increases the size of the `queue` by one. The new element is created by performing a copy of parameter `x`. This method is a proxy, which means it calls the `push_back()` method in the underlying storage container.

Name: pop
Signature:
```
void pop ( );
```
Parameters:
None
Return Value: None
Description:

The pop() method removes the first element from the queue and therefore reduces its size by one. If the queue is not empty, the next element (previously second in the sequence) becomes the new front of the queue.

It's important to remember that the pop() call doesn't return the removed element. If you want to retrieve the value of the front of the queue, call the front() method instead. This method is a proxy, which means it calls the pop_front() method in the underlying storage container.

File name: 1.5.12.cpp

```
#include <queue>
#include <iostream>

using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};
    queue<int> q1;
    q1.push(100);
    cout<<"Front: "<<q1.front() << " Back: "<<q1.back()<<endl;
    q1.push(13);
    cout<<"Front: "<<q1.front() << " Back: "<<q1.back()<<endl;
    q1.push(44);
    cout<<"Front: "<<q1.front() << " Back: "<<q1.back()<<endl;

    q1.pop();
    cout<<"Front: "<<q1.front() << " Back: "<<q1.back()<<endl;
    return 0;
}
```

1.5.14 Priority queue

Name: <queue>
Signature:
```
template < class T, class Container = vector<T>,\ class Compare = less<typename Container::value_ty
```
Parameters:

T – the type of the elements stored inside the priority_queue
Container – the internal storage provider
Compare – the comparator used for ascertaining the strict weak ordering inside the priority_queue ;

`Container::value_type` – the type of the elements stored inside the `priority_queue` – it holds the same meaning as `T`.

Return Value: None

Description:

`priority_queue` is a data structure in which the greatest element is always the first one, due to some predefined strict weak ordering condition. The STL `priority_queue` is just an implementation of that principle.

In order for `priority_queue` to function, it requires a container to provide internal storage. This container must offer the following functionality and methods: the elements must be accessible through the random access iterator; `front(); push_back(); pop_back();`

Therefore, only `vector` and `deque` of the STL containers are applicable. But again, any container supporting the required functionality will be feasible; a programmer is not limited to just the STL containers. If the internal storage type is not specified, a `vector` is used.

`priority_queue` maintains its order using a comparator. The comparator is a function which takes two parameters and returns true if the first parameter is lower than the second due to some strict weak ordering. It can be implemented as either a functional object, or a standalone method. This comparator is used when a new element is inserted (pushed) into the `queue`, or removed (popped) from the `queue`, to guarantee that the element on top of it is always the greatest. If no comparator is specified during the instantiation of the `priority_queue` – less, the STL predicate is used.

The ordered sequence of elements is achieved by using the `heap` algorithm. `priority_queue` uses the following methods (algorithms): `make_heap(), push_heap(), pop_heap()` .

1.5.15 Priority queue – initialization

Name: `constructor`

Signature:

```
explicit priority_queue ( const Compare& x = Compare(),\ const Container& y = Container() );\ priority
```

Parameters:

`x` – the comparator object used to ensure strict weak ordering of the `queue` – must be of the same type as declared in the template parameters;

`y` – the container provided to serve as internal storage. Its type must be the same as the type defined in the `priority_queue` template declaration;

`other` – the already existing `priority_queue` , which is used as a template to instantiate the current object; its internal container must be of the same type as the object being created. Also, both objects must use the same comparator.

Return Value: None

Description:

First, the constructor creates a `priority_queue` using the internal container provided, and compares the predicate. The container does not have to be empty; there might be objects already inside. However, the cont needs to be of the same type as specified in the `priority_queue` object declaration – the Container template parameter. If objects `x` and `y` are provided, their copies are created; in other cases, the container and comparator are just initialized. After that, the constructor calls `make_heap()` .

Second, the constructor is just a copy constructor. It creates a new object by coping the existing one. Both objects must be equal in terms of their template definition – the same type of container and comparator predicates.

File name: 1.5.14.cpp

```
#include <iostream>
#include <queue>
#include <deque>
#include <vector>
#include <functional>

using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};

    priority_queue<int>  s1;

    priority_queue<int>  s2(s1);

    vector<int>  v1(a1, a1+10);
    priority_queue<int>  s3(v1.begin(), v1.end());

    priority_queue<int, deque <int> > s4;

    priority_queue<int, vector<int>, greater <int> > s5;
    return 0;
}
```

1.5.16 Priority queue – initialization

Priority queue initialization – the wrong way

- 1. using external storage object only
- 2. using forbidden container type as internal storage
- 3. providing comparator but not container type
- 4. using different comparator object than declared - warning, but the comparator object type is deducted from constructor parameter

File name: 1.5.15.cpp

```
#include <deque>
#include <list>
#include <vector>
```

```
#include <queue>
#include <functional>

using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};
    vector<int>    v1(a1,a1+10);

    priority_queue<int>  q1(v1);

    priority_queue<int, list <int> > q2;

    priority_queue<int, greater <int> > q3;

    priority_queue<int>  q4(greater<int>());
    return 0;
}
```

1.5.17 Priority queue – assignment operator

Name: `assignment operator =`
Signature:
`priority_queue<T, Container>&\ operator=( const priority_queue<T,Container>& other );`
Parameters:

`other` – the source `priority_queue` object, whose contents are copied and stored in a target; `priority_queue` . The source and target objects must use the same type of internal container.

Return Value: A reference to a calling object (*this).
Description:

The assignment operator copies the elements from other, and places them in a calling object (l-value). As in the case of the copy constructor, the source and target objects must be created using the same type of internal storage.

1. Correct - source and target stack are of the same type

2a. Incorrect - target and source queue are not of the same type - internal storage

2b. Incorrect - target and source queue are not using the same comparator

File name: 1.5.16.cpp

```
#include <iostream>
#include <queue>
#include <deque>
#include <vector>
#include <functional>

using namespace std;
```

```
int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};

    priority_queue<int>  q1(a1, a1+10);
    priority_queue<int>  q2;

    q1 = q2;
    cout<<q1.size()<<":"<<q2.size()<<endl;

    priority_queue<int, deque <int> > q3;

    q3 = q2;
    priority_queue<int, vector <int>, greater <int> > q4;

    q4 = q1;
    return 0;
}
```

1.5.18 Priority queue – empty() and size() methods

Name: `empty`
Signature:
`bool empty ( ) const;`
Parameters:
None
Return Value: `true` if the `priority_queue` size is 0, and `false` otherwise.
Description:
This method tests if the `priority_queue` size is 0, which means the `queue` is empty. The function is just a proxy which calls the method of the same name in the underlying container.
Name: `size`
Signature:
`size_type size ( ) const;`
Parameters:
None
Return Value: The number of elements currently stored inside a `priority_queue`
Description:
Returns the number of elements stored in a `priority_queue` . The function is just a proxy which calls the method of the same name in the underlying container.
File name: 1.5.17.cpp

```
#include <queue>
#include <iostream>
```

```
using namespace std;

int main()
{
    priority_queue<int> q1;
    if (q1.empty())
    {
        q1.push(100);
    }
    cout<<"Size of queue: " << q1.size() << endl;
    return 0;
}
```

1.5.19 Priority queue – top(), push() and pop()

Name: `top`
Signature:
`const value_type& top ( ) const;`
Parameters:
None
Return Value: A constant reference to the element at the top of a `priority_queue`
Description:

This function returns a reference to the top-most element of a `priority_queue` . Because a `priority_queue` uses strict weak ordering, the top element is the greatest one. It's also worth noticing that the returned reference is a `const`. There's no non-const version of this method, as in a `stack`, for example. This is because of the ordered nature of the `priority_queue` . A reference would allow for unnoticeable changes of element values, and would disrupt the order of the `queue`.

`top()` is a proxy method, which means it calls the `front()` method of the underlying container. The member type `value_type` is defined as the type of the elements from the underlying container.

Name: `push`
Signature:
`void push ( const T& x );`
Parameters:
`x` - the value to be pushed (copied) into the `queue`;
`T` – this is the first template parameter – the type of the elements stored inside the container.
Return Value: None
Description:

The method `push()` puts a new value into the `priority_queue` , and rearranges it to keep the proper heap structure. This effectively increases the size of the adaptor by one. The location of the newly inserted element is not determined beforehand; it depends on the values of the elements already stored inside the `queue`. The new element is created by copying parameter `x`. This method is a proxy, which means it calls the `push_back()` method in the underlying storage container. Afterward, the `push_heap()` method is called.

Name: `pop`
Signature:

```
void pop ( );
```

Parameters:

None

Return Value: None

Description:

The `pop()` method removes the top-most element from the `priority_queue` , and therefore reduces its size by one, and the element destructor is called in the process. Also, this method ensures that the new top element is the greatest one out of all the remaining elements. It's important to remember that the `pop()` call doesn't return the removed element.

If you want to retrieve the value of the top of the `queue`, call the `top()` method. This method is a proxy, which means it calls the `pop_heap()` algorithm, and then `pop_back()` in the underlying storage container.

File name: 1.5.18.cpp

```
#include <iostream>
#include <queue>
#include <deque>
#include <vector>
#include <functional>

using namespace std;

int main()
{
    int a1[]={1, 100, 34, 23, 9};

    priority_queue<int>  q1(a1, a1+5);
    priority_queue<int, deque<int>, greater <int> > q2(a1, a1+5);
    cout<< "q1 queue top: " << q1.top()<<endl;
    cout<< "q2 queue top: " << q2.top()<<endl;
    cout<<"Adding value: 101\n";
    q1.push(101);
    q2.push(101);
    cout<< "q1 queue top: " << q1.top()<<endl;
    cout<< "q2 queue top: " << q2.top()<<endl;
    cout<<"Removing top of the queue: \n";
    q1.pop();
    q2.pop();
    cout<< "q1 queue top: " << q1.top()<<endl;
    cout<< "q2 queue top: " << q2.top()<<endl;

    return 0;
}
```

Chapter 2

Associative STL containers

2.1 Introduction

`Associative containers` are containers in which every element stored inside is accessible by a special value called a **key**.

Therefore, there is an association between the element and the key. In `sequence containers`, elements are accessed by their place (index) in the container.

Associative containers are usually implemented on the basis of **binary search tree algorithms**, or even balanced binary trees. This makes them efficient storage containers, but usually quite big ones in comparison to other collections. In some rare cases this might be an issue.

There are **four** types of `associative containers` in the STL:

- `set`
- `multiset`
- `map`
- `multimap`

In fact, there is the **bitset** too, but we'll skip it in this tutorial. As you'll see in the table on the following slide, all associative containers have a very similar interface, yet there are some fundamental differences between the `set` and `map` concepts.

The table shows the most important operations provided by STL associative containers. As you can see, the majority of the methods are common between the different types of containers.

2.2 Set and multiset

2.2.1 Template definition

Below you can see the template definition of the `set` and `multiset` classes. Both classes are located in the same header file. **Name:** `<set>`

Signature:

```
template < class Key, class Compare = less<Key>, class Allocator = allocator<Key> > class set; temp
```

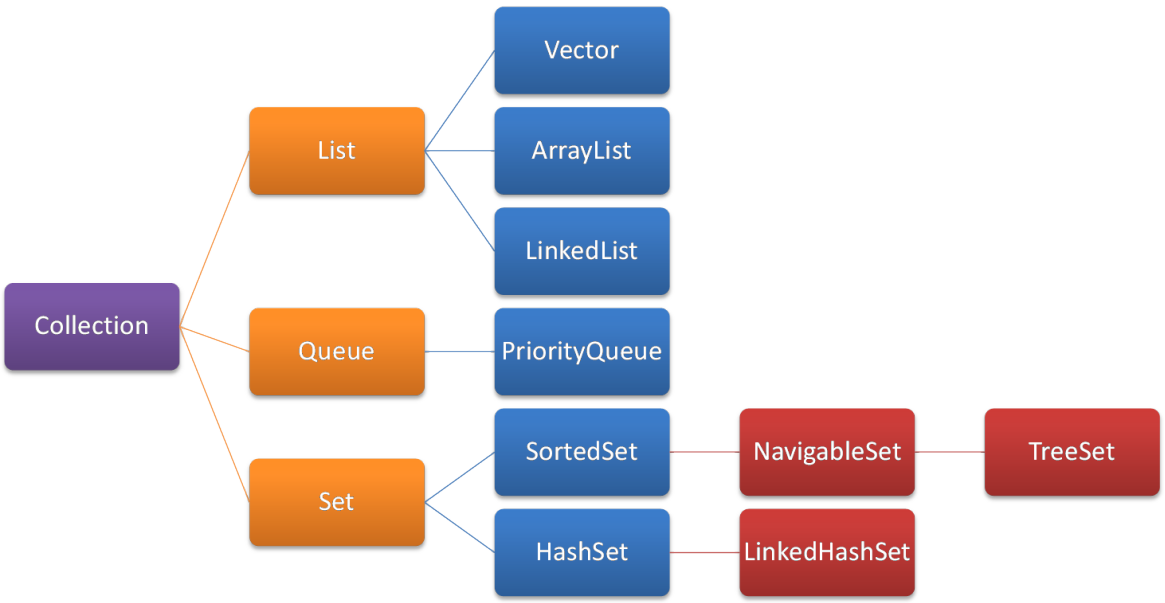

Fig. 2.1 Function table

Parameters:

`Key` – the type of key stored inside the set, and therefore the type of the elements themselves

`Compare` – the type of comparator used to perform a comparison between the `set` elements, in order to ensure strict weak ordering. It can be implemented as a two-argument function, or a functional object. If no comparator is provided, `less()` will be used as a default;

`Allocator` – the type of allocator used to provide the storage allocation model.

Return Value: None

Description:

`Set` and `multiset` are associative containers in which the elements stored inside them are `keys` themselves.

2.2.2 Set and multiset functionality

All `set` features are grouped in the table on the slide.

As you can see, there's only one difference between these two classes, and this is why they'll be described together. We'll simply refer to both containers as `set` in the rest of this tutorial, with some exceptions, when their functionality differs.

2.2.3 Strict weak ordering & the comparator object

The elements inside the `set` are sorted using a strict weak ordering algorithm. In order for that to happen, two consecutive elements must be compared to one another. The comparison is performed using the second template parameter – `Compare`.

As you can see, the default value of that argument is `less<Key>`. In order for it to work, `less<Key>` requires the operator `<` to be defined for the type `Key`. If a particular type does not support this operation, a custom comparator must be created and passed during the `set` object definition.

Another reason to create a custom comparator is to provide some specific ordering.

	set	multiset
Key and element identity	Key and element are one and the same object.	
Uniqueness	No duplicates are allowed inside the set – it is not possible to insert two elements with the same value into the set.	Duplicated values are allowed, you can insert duplicate values into multiset.
Ordering	All elements inside the set are sorted using strict weak ordering concept. The order in a particular set container depends on the Compare parameter of the template. less() is the default one in which case elements are sorted in an ascending order.	
Key immutability	The value of the key cannot be changed. As the key and element are both the same object, it means you cannot update an element which has been put into the set.	

Fig. 2.2 Function table

A comparator `object` is required for the `set` and `multiset` to properly order the elements.

On the slide, you can see two possible prototypes of a comparator. The comparator should return `true` if the element k1 is placed before the element k2. Using a functional object seems to be more feasible, as you need to provide the comparator type as a template parameter Compare during the `set` object instantiation. If you intend to pass arguments to the comparator by reference, those references must be `const` – a key immutability restriction.

`sets` are usually preferred when you want to make sure that there's no duplication of elements inside the container. The biggest issue with STL `set` implementation is that there's no way to avoid ordering, which sometimes might complicate the use case; especially when we intend to put objects into a `set`.

You might need to provide a custom comparator, or modify a custom class by overloading the < (smaller than) operator, which will allow `less()` to deal with these objects.

File name: 2.2.3.cpp

```
template<class Key>
bool cmp(Key k1, Key k2);

template<class Key>
struct CMP{
  bool operator()(Key k1, Key  k2);
};
```

2.2.4 Set and multiset operations

- constructor
- destructor
- operator=

- begin
- end
- rbegin
- rend
- empty
- size
- max_size
- insert
- erase
- clear
- swap
- find
- count
- lower_bound
- upper_bound
- equal_range

2.2.5 Constructors and destructors

Name:

`constructor`

Signatures:

```
explicit set ( const Compare& comp = Compare(), const Allocator& = Allocator() );
template <class InputIterator>
set ( InputIterator first, InputIterator last, const Compare& comp = Compare(), const Allocator& = All
set ( const set<Key,Compare,Allocator>& x );
explicit multiset ( const Compare& comp = Compare(), const Allocator& = Allocator() );
template <class InputIterator>
multiset ( InputIterator first, InputIterator last, const Compare& comp = Compare(), const Allocator&
multiset ( const multiset<Key,Compare,Allocator>& x );
```

Parameters:

- `first, last` – the iterators specifying the range of elements to be inserted into the container during object creation. As usual, the range includes first and excludes last;

- `comp` – the comparator to be used in order to provide the strict weak ordering of the `set`. Its type must be the same as that specified by the template parameter Compare. If this argument is omitted, the default value will be used;
- `unnamed allocator` – the allocator object to be used;
- `x` – the already existing set object; its template parameters must be set to the same values as the object being created.

Return Value: None
Description:

- The first constructor simply creates an empty `set` object using the optional parameters `comp` and `unnamed allocator`, if they're provided. If these arguments are not supplied, this constructor becomes the default constructor.
- The second constructor creates a set and fills it with the elements provided by another collection. The iterators `first` and `last` define the range of elements used during this initialization.
- The third constructor is just a copy constructor. The newly created object will be an exact copy of the existing one. Both objects must be declared with the same values of the template parameters.

Name: `destructor`
Signature:
`~set(); ~multiset();`
Parameters:
None
Return Value: None
Description:
It destroys the `set` container object. The destructors of all the stored objects are called, and the whole allocated storage is released.
File name: 2.2.5.cpp

```
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        while (start != end) {
                std::cout << *start << " "; start++;
        }
    }

int main()
{
```

```
    int t[]={1, 10, 8, 4, 6, 5, 3, 9, 7, 2};
    set <int> s1;
    set <int> s2(t, t+10);
    print(s2.begin(), s2.end()); cout<<endl;
    set <int> s3(s2);
    print(s3.begin(), s3.end()); cout<<endl;

    set<int, greater<int> > s4 (t,t+10);
    print(s4.begin(), s4.end()); cout<<endl;

    set <int> s6;
    s6 = s3;
    print(s6.begin(), s6.end()); cout<<endl;
    return 0;
}
```

```
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        while (start != end) {
                std::cout << *start << " "; start++;
        }
    }


int main()
{
    int t[]={2, 10, 8, 4, 5, 5, 3, 10, 7, 2};
    multiset <int> s1;
    multiset <int> s2(t, t+10);
    print(s2.begin(), s2.end()); cout<<endl;
    multiset <int> s3(s2);
    print(s3.begin(), s3.end()); cout<<endl;

    multiset<int, greater <int> > s4 (t,t+10);
    print(s4.begin(), s4.end()); cout<<endl;
```

```
    multiset <int> s6;
    s6 = s3;
    print(s6.begin(), s6.end()); cout<<endl;
    return 0;
}
```

2.2.6 Assignment operator

Name: `operator=`

Signature:

```
set<Key,Compare,Allocator>& operator= ( const set<Key,Compare,Allocator>& x ); multiset<Key,Compare
```

Parameters:

`x` – the set object used as the source for the assignment operation;

`Key, Compare, Allocator` – these parameters are the template parameters of the `set/multiset` class.

Return Value: A reference to itself `(*this)`.

Description:

Copies the contents of the source object to the target object. After this operation, both objects are identical. The source and target objects must be defined with the same values as the template parameters.

File name: 2.2.6.cpp

```
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        while (start != end) {
                std::cout << *start << " "; start++;
        }
    }

int main()
{
    int t[]={1, 10, 8, 4, 6, 5, 3, 9, 7, 2};

    set <int> s1;

    set <int> s2(t, t+10);
    print(s2.begin(), s2.end()); cout<<endl;

    set <int> s3(s2);
    print(s3.begin(), s3.end()); cout<<endl;
```

```
    set<int, greater <int> > s4 (t,t+10);
    print(s4.begin(), s4.end()); cout<<endl;

    set <int> s6;
    s6 = s3;
    print(s6.begin(), s6.end()); cout<<endl;

    return 0;
}


#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        while (start != end) {
                std::cout << *start << " "; start++;
        }
    }


int main()
{
    int t[]={2, 10, 8, 4, 5, 5, 3, 10, 7, 2};

    multiset <int> s1;

    multiset <int> s2(t, t+10);
    print(s2.begin(), s2.end()); cout<<endl;

    multiset <int> s3(s2);
    print(s3.begin(), s3.end()); cout<<endl;

    multiset<int, greater <int> > s4 (t,t+10);
    print(s4.begin(), s4.end()); cout<<endl;

    multiset <int> s6;
    s6 = s3;
    print(s6.begin(), s6.end()); cout<<endl;

    return 0;
```

```
}
```

2.2.7 Iterator methods

Name: `begin`
Signature:
```
iterator begin (); const_iterator begin () const;
```
Parameters:
None
Return Value: The iterator which points to the first element in the `set`
Description:
This method returns an iterator which points to the first element (the key) of the `set`. The function comes in two variants: const and non-const. It's important to remember that due to the key invariability of the set, there's no functional difference between them.
Name: `end`
Signature:
```
iterator end(); const_iterator end() const;
```
Parameters:
None
Return Value: The iterator to the past-the-end element of the `set`
Description:
This method returns an iterator which points to the past-the-end element (the `key`) of the `set`. The past-the-end element is a virtual element located after the last element of the `set`. It indicates the end of the `set`. The function comes in two variants: `const` and `non-const`. It's important to remember that due to the key invariability of the `set`, there's no functional difference between them.
Name: `rbegin`
Signature:
```
reverse_iterator rbegin (); const_reverse_iterator rbegin () const;
```
Parameters:
None
Return Value: The reverse iterator which points to the last element in the `set(key)`
Description:
This method returns a reverse iterator which points to the last element (the key) of the `set`. Reverse iterators iterate through the collections in reverse order – from the end to the start. The function comes in two variants: const and non-const. It's important to remember that due to the key invariability of the `set`, there's no functional difference between them.
Name: `rend`
Signature:
```
reverse_iterator rend(); const_reverse_iterator rend() const;
```
Parameters:
None
Return Value: The reverse iterator of the virtual element located before the first real element of the `set`
Description:
This method returns an iterator which points to the virtual element located before the first real element (the

key) of the set. It indicates the end of the set in reverse order - from the to the start. The function comes in two variants: const and non-const. It's important to remember that due to the key invariability of the set, there's no functional difference between them.

File name: 2.2.7.cpp

```
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }

int main()
{
    int t[]={1, 10, 8, 4, 6, 5, 3, 9, 7, 2};
    set <int> s1(t, t+10);
    set<int, greater <int> > s2(s1.begin(), s1.end());
    cout<<"Standard iterator:\n";
    set<int>::iterator it1 = s1.begin();
    for( ; it1 != s1.end(); ++it1)
    {
        cout<<*it1<<" ";
    }
    cout<<endl;
    it1 = s2.begin();
    for( ; it1 != s2.end(); ++it1)
    {
        cout<<*it1<<" ";
    }
    cout<<endl;
    cout<<"Reverse iterators:\n";
    print(s1.rbegin(), s1.rend()); cout<<endl;
    print(s2.rbegin(), s2.rend()); cout<<endl;

    cout<<"Const iterators: \n";
    print(s1.cbegin(), s1.cend()); cout<<endl;
    print(s2.cbegin(), s2.cend()); cout<<endl;
    cout<<"Const iterators - reverse: \n";
    print(s1.crbegin(), s1.crend()); cout<<endl;
    print(s2.crbegin(), s2.crend()); cout<<endl;
```

```
    set<int>::const_iterator cit1 = s1.begin();
    it1 = s1.cbegin();
    it1 = s1.begin();
    return 0;
}
```

```
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }

int main()
{
    int t[]={1, 10, 8, 4, 6, 5, 3, 9, 7, 2};
    multiset <int> s1(t, t+10);
    multiset<int, greater <int> > s2(s1.begin(), s1.end());
    cout<<"Standard iterator:\n";
    multiset<int>::iterator it1 = s1.begin();
    for( ; it1 != s1.end(); ++it1)
    {
        cout<<*it1<<" ";
    }
    cout<<endl;
    it1 = s2.begin();
    for( ; it1 != s2.end(); ++it1)
    {
        cout<<*it1<<" ";
    }
    cout<<endl;
    cout<<"Reverse iterators:\n";
    print(s1.rbegin(), s1.rend()); cout<<endl;
    print(s2.rbegin(), s2.rend()); cout<<endl;
```

```
    cout<<"Const iterators: \n";
    print(s1.cbegin(), s1.cend()); cout<<endl;
    print(s2.cbegin(), s2.cend()); cout<<endl;
    cout<<"Const iterators - reverse: \n";
    print(s1.crbegin(), s1.crend()); cout<<endl;
    print(s2.crbegin(), s2.crend()); cout<<endl;


    multiset<int>::const_iterator cit1 = s1.begin();
    it1 = s1.cbegin();
    it1 = s1.begin();
    return 0;
}
```

2.2.8 Size-related methods

Name: `empty`
Signature:
`bool empty () const;`
Parameters:
None
Return Value: This method returns `true` if the set is empty and `false` otherwise.
Description:
This method is used to indicate whether the `set` is empty or not.
Name: `size`
Signature:
`size_type size() const;`
Parameters:
None
Return Value: The number of elements which are currently stored inside the `set`.
Description:
This method returns the number of elements which are currently stored inside the `set`. The size will change each time an element is added to or removed from the `set`.
Name: `max_size`
Signature:
`size_type max_size () const;`
Parameters:
None
Return Value: The maximum number of elements which can be held inside the `set`.
Description:
This method returns the maximum physical capacity of the `set`. This value might depend on the STL library implementation or operating system, and will always be constant in the same environment.
File name: 2.2.8.cpp

```
#include <set>
```

```
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }

int main()
{
    int t[]={2, 10, 8, 4, 5, 5, 3, 10, 7, 2};
    multiset <int> s1(t, t+10);
    set <int> s2(s1.begin(), s1.end());
    cout<<"Multiset:\n";
    cout<<"Size: "<< s1.size()<< " Max size: " <<s1.max_size()<<endl;
    print(s1.begin(), s1.end()); cout<<endl;
    cout<<"Set:\n";
    cout<<"Size: "<< s2.size()<< " Max size: " <<s2.max_size()<<endl;
    print(s2.begin(), s2.end()); cout<<endl;

    cout<<"Deleting all elements from the multiset\n";
    s1.clear();
    if (s1.empty())
    {
        cout<<"Multiset is empty!\n";
    }
    else
    {
        print(s1.begin(), s1.end()); cout<<endl;
    }

    if (s2.empty())
    {
        cout<<"Set is empty!\n";
    }
    else
    {
        print(s2.begin(), s2.end()); cout<<endl;
    }
    return 0;
}
```

2.2.9 Insert method

Name: `insert`
Signature:
`pair<iterator, bool> insert(const key_type& x ); iterator insert (iterator position, const key_type& x`
Parameters:

`x`: the value to be inserted into the `set`

`position`: the position at which value `x` should be inserted – the insertion point, if chosen properly, can result in some optimization. Nonetheless, element `x` will be inserted into the position which follows the existing order of the `set`.

`first, last`: the iterators specifying the range of elements to be inserted into the container during object creation. As usual, the range includes `first` and excludes `last`.

Return Value: The first version (`set`) returns a structure `pair` in which the `first` field is an iterator of the newly inserted element, or an already existing element in the `set`. The information stating which scenario has happened is stored inside the second field: `true` – insertion, `false` – a value already exists.

the second version (`set`) returns an iterator which points either to the newly inserted element, or to an already existing element.

for `multiset`, the returned value always points to the newly inserted element.

Description:

The function `insert()` is used to put new values into a `set`. Each successful call of this method will effectively increase the size of the `set`. As the `set` container does not allow duplicate values, each element to be inserted is checked to make sure that it doesn't violate this condition. If it does, the insertion will be unsuccessful.

As `multiset` allows for duplicate elements, there can never be an unsuccessful scenario for a call to `insert()`.

Another restriction is related to the order of the elements in the `set`. The new element will be placed at a position which follows the set-ordering policy. As we stated earlier, the parameter `position` in the second version of this method has only an informative meaning, suggesting a possible insertion point, but can result in performance improvements during the insertion process. A newly inserted element is obtained by creating a copy of parameter `x`.

The last version of `insert()` inserts elements from range defined by the iterators `first` and `last`. All these restrictions still apply for both classes respectively.

File name: 2.2.9.cpp

```
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }
```

```
typedef pair<set<int>::iterator, bool> Pair;

void check(const Pair & result)
{
    if (result.second == true)
    {
        cout<<"A new value ("<<*result.first<< ") has been inserted"<<endl;
    }
    else
    {
        cout<<"Insertion failed, value "<< *result.first<<" already exists\n";
    }
}

int main()
{
    int t[]={16, 10, 8, 40, 6, 15, 3, 9, 7, 2};
    set <int> s1(t, t+10);
    cout<<"The first version of insert:\n";

    print(s1.begin(), s1.end()); cout<<endl;
    Pair p = s1.insert(10);
    check(p);
    print(s1.begin(), s1.end()); cout<<endl;

    p = s1.insert(13);
    check(p);
    print(s1.begin(), s1.end()); cout<<endl<<endl;
    cout<<"The second version of insert:\n";
    set<int>::iterator it1 = s1.insert(s1.find(10),11);
    set<int>::iterator it2 = s1.insert(s1.find(11),11);
    if (it1 == it2)
    {
        cout<<"Second insertion was not successful\n";
    }
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"The third version of insert:\n";
    int t2[]={4,10,15,21,0};
    s1.insert(t2, t2+5);
    print(s1.begin(), s1.end()); cout<<endl;

    return 0;
```

```
}
```

```
#include <set>
#include <iostream>
#include <functional>
#include <iterator>

using namespace std;

template<class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }


int main()
{
    int t[]={16, 10, 8, 40, 6, 15, 3, 9, 7, 2};
    multiset <int> s1(t, t+10);


    print(s1.begin(), s1.end()); cout<<endl;
    cout<<"The first version of insert:\n";
    multiset<int>::iterator it1 = s1.insert(13);
    print(s1.begin(), s1.end()); cout<<endl;
    cout<<"Inserted element position: "<<distance(s1.begin(), it1)<<endl<<endl;

    multiset<int>::iterator it2 = s1.insert(13);
    print(s1.begin(), s1.end()); cout<<endl;
    cout<<"Inserted element position: "<<distance(s1.begin(), it2)<<endl<<endl;

    cout<<"The second version of insert:\n";
    it1 = s1.insert(s1.find(10),11);
    it2 = s1.insert(s1.find(11),11);
    if (it1 == it2)
    {
        cout<<"Second insertion was not successful\n";
    }
    print(s1.begin(), s1.end()); cout<<endl<<endl;
```

```
    cout<<"The third version of insert:\n";
    int t2[]={4,10,15,21,0};
    s1.insert(t2, t2+5);
    print(s1.begin(), s1.end()); cout<<endl;

    return 0;
}

set

pair<iterator,bool> insert ( const value_type& x );
iterator insert ( iterator position, const value_type& x );
template <class InputIterator>
void insert ( InputIterator first, InputIterator last );

multiset
iterator insert ( const value_type& x );
iterator insert ( iterator position, const value_type& x );
template <class InputIterator>
void insert ( InputIterator first, InputIterator last );
```

2.2.10 Erase methods

Erase methods: removing elements from the container. **Name:** `erase`

Signature:

```
void erase ( iterator position ); size_type erase ( const key_type& x ); void erase ( iterator firs
```

Parameters:

`position`: the iterator pointing to the element to be removed.

`x`: the element (`key`) to be removed from the `set`, `key_type` is an alias to `Key`, which is the `first` template parameter.

`first, last`: the iterators specifying the range of elements to be removed from the `set`. As usual, the range includes first and excludes last.

Return Value: The number of elements removed: for the `set`, it is either 1 if value `x` has been removed, or 0 otherwise; for a `multiset`, the function returns the number of times value `x` is presented in the `multiset`, or 0 if it isn't found at all.

Description:

This function removes elements from the collection. As a result, the `set` size will be decreased and the destructor of each deleted element will be called. The deletion might be performed using one of three possible methods:

iterator

key value

range of iterators

File name: 2.2.10.cpp

```
#include <set>
```

```
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }

int main()
{
    int t[]={16, 10, 8, 40, 6, 15, 3, 9, 7, 2};
    set<int, greater<int> > s1(t,t+10);
    print(s1.begin(), s1.end()); cout<<endl<<endl;
    cout<<"Removing element from a certain position (iterator):\n";
    set<int>::iterator it = s1.find(15);
    s1.erase(it);
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"Removing certain value (9) from the set:\n";
    s1.erase(9);
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"Removing range of iterators:\n";
    s1.erase(s1.find(6), s1.end());
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"Removing all the elements from the set\n";
    s1.clear();
    cout<<"S1 size: "<<s1.size()<<endl;
    return 0;
}

#include <set>
#include <iostream>
#include <multiset>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
```

```
                std::cout << *start << " ";
        }
    }

int main()
{
    int t[]={9, 10, 8, 15, 6, 15, 3, 9, 7, 2 };
    multiset<int, greater > s1(t,t+10);
    print(s1.begin(), s1.end()); cout<<endl<<endl;
    cout<<"Removing element from a certain position (iterator):\n";
    multiset::iterator <int> it = s1.find(15);
    s1.erase(it);
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"Removing certain value (9) from the multiset:\n";
    s1.erase(9);
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"Removing range of iterators:\n";
    s1.erase(s1.find(6), s1.end());
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"Removing all the elements from the set\n";
    s1.clear();
    cout<<"S1 size: "<<s1.size()<<endl;
    return 0;
}
```

2.2.11 swap method

Name: `swap`
Signature:

```
void swap ( set<Key,Compare,Allocator>& st );
```

Parameters:

`st`: the other set container identical in meaning to the template parameters with this method's caller.

Return Value: None

Description:

This function performs the exchange of all contents between two set containers. After the call, all the elements stored in the set will be placed in st and vice versa. All iterators, references and pointers obtained before the call are still valid, but in relation to the swapped set objects. No construction or destruction of elements inside either of the sets is performed during the call.

File name: 2.2.11.cpp

```
#include <set>
#include <string>
```

```
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }

int main()
{
    string t1[]={"aaa", "bbb", "ccc"};
    string t2[]={"xxx", "yyy", "zzz"};
    set <string> s1(t1, t1+3);
    set <string> s2(t2, t2+3);
    cout<<"S1: ";     print(s1.begin(), s1.end());cout<<endl;
    cout<<"S2: ";   print(s2.begin(), s2.end());cout<<endl;
    cout<<"Swap:\n";
    s1.swap(s2);
    cout<<"S1: ";     print(s1.begin(), s1.end());cout<<endl;
    cout<<"S2: ";   print(s2.begin(), s2.end());cout<<endl;
    cout<<"Swap:\n";
    s2.swap(s1);
    cout<<"S1: ";     print(s1.begin(), s1.end());cout<<endl;
    cout<<"S2: ";   print(s2.begin(), s2.end());cout<<endl;
    return 0;
}

#include <set>
#include <string>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }

int main()
```

```
{
    string t1[]={"aaa", "bbb", "ccc"};
    string t2[]={"xxx", "yyy", "zzz"};
    multiset <string> s1(t1, t1+3);
    multiset <string> s2(t2, t2+3);
    cout<<"S1: ";     print(s1.begin(), s1.end());cout<<endl;
    cout<<"S2: ";   print(s2.begin(), s2.end());cout<<endl;
    cout<<"Swap:\n";
    s1.swap(s2);
    cout<<"S1: ";     print(s1.begin(), s1.end());cout<<endl;
    cout<<"S2: ";   print(s2.begin(), s2.end());cout<<endl;
    cout<<"Swap:\n";
    s2.swap(s1);
    cout<<"S1: ";     print(s1.begin(), s1.end());cout<<endl;
    cout<<"S2: ";   print(s2.begin(), s2.end());cout<<endl;
    return 0;
}
```

2.2.12 find method

Name: `find`

Signature:

```
iterator find ( const key_type& x ) const;
```

Parameters:

`x`: the value to be searched for in the set.

Return Value: If successful, an iterator to the found element is returned. If the value cannot be found, the function returns `set::end()` (past-the-end element)

Description:

The function `find()` looks for value x inside the set. If the value is found, the iterator to it is returned. A search failure is indicated by returning the `set::end()` value.

For a `multiset`, this function returns an iterator pointing to the first place in which this value is present inside the container.

File name: 2.2.12.cpp

```
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }
```

```
typedef pair<set<int>::iterator, bool> Pair;

void check(const Pair & result)
{
    if (result.second == true)
    {
        cout<<"A new value ("<<*result.first<< ") has been inserted"<<endl;
    }
    else
    {
        cout<<"Insertion failed, value "<< *result.first<<" already exists\n";
    }
}

int main()
{
    int t[]={16, 10, 8, 40, 6, 15, 3, 9, 7, 2};
    set <int> s1(t, t+10);
    cout<<"The first version of insert:\n";

    print(s1.begin(), s1.end()); cout<<endl;
    Pair p = s1.insert(10);
    check(p);
    print(s1.begin(), s1.end()); cout<<endl;

    p = s1.insert(13);
    check(p);
    print(s1.begin(), s1.end()); cout<<endl<<endl;
    cout<<"The second version of insert:\n";
    set<int>::iterator it1 = s1.insert(s1.find(10),11);
    set<int>::iterator it2 = s1.insert(s1.find(11),11);
    if (it1 == it2)
    {
        cout<<"Second insertion was not successful\n";
    }
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"The third version of insert:\n";
    int t2[]={4,10,15,21,0};
    s1.insert(t2, t2+5);
    print(s1.begin(), s1.end()); cout<<endl;

    return 0;
```

```
}

#include <set>
#include <iostream>
#include <functional>
#include <iterator>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }


int main()
{
    int t[]={16, 10, 8, 40, 6, 15, 3, 9, 7, 2};
    multiset <int> s1(t, t+10);


    print(s1.begin(), s1.end()); cout<<endl;
    cout<<"The first version of insert:\n";
    multiset<int>::iterator it1 = s1.insert(13);
    print(s1.begin(), s1.end()); cout<<endl;
    cout<<"Inserted element position: "<<distance(s1.begin(), it1)<<endl<<endl;

    multiset<int>::iterator it2 = s1.insert(13);
    print(s1.begin(), s1.end()); cout<<endl;
    cout<<"Inserted element position: "<<distance(s1.begin(), it2)<<endl<<endl;

    cout<<"The second version of insert:\n";
    it1 = s1.insert(s1.find(10),11);
    it2 = s1.insert(s1.find(11),11);
    if (it1 == it2)
    {
        cout<<"Second insertion was not successful\n";
    }
    print(s1.begin(), s1.end()); cout<<endl<<endl;

    cout<<"The third version of insert:\n";
    int t2[]={4,10,15,21,0};
    s1.insert(t2, t2+5);
```

```
    print(s1.begin(), s1.end()); cout<<endl;

    return 0;
}
```

2.2.13 Count method

Name: `count`
Signature:
`size_type count ( const key_type& x ) const;`
Parameters:
`x`: the value to be looked for in the set.
Return Value: Returns the number of times value `x` has been found in the `set/multiset`. In the case of a `set`, it either returns 1 if the searched value has been found, or 0 otherwise.
Description:
The function `count()` looks for value `x` and returns the number of times this value occurs in the `set/multiset`. Since the set does not allow for duplicates, this means that the function returns either 1 or 0.

For a `multiset`, the returned number can, of course, be greater than 1.

File name: 2.2.13.cpp

```
#include <set>
#include <string>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }

int main()
{
    int t[]={2, 10, 8, 4, 5, 5, 3, 10, 7, 2};
    set<int, greater<int> > s1(t,t+10);
    multiset<int, greater<int> > s2(t,t+10);
    cout<<"S1: ";     print(s1.begin(), s1.end());cout<<endl;
    cout<<"S2: ";   print(s2.begin(), s2.end());cout<<endl;
    cout<<"How many '2' there are?"<<endl;
    cout<<"S1: "<< s1.count(2)<<endl;
    cout<<"S2: "<< s2.count(2)<<endl;
    if (s1.count(1) >0)
    {
```

```
        cout<<"There is '1' in the set\n";
    }
    else
    {
        cout<<"There is no '1' in the set\n";
    }
    return 0;
}
```

2.2.14 Bounds related methods

Name: `lower_bound`
Signature:
`iterator lower_bound ( const key_type& x ) const;`
Parameters:
`x`: the value to be looked for inside the set.
Return Value: The iterator to the first element which is greater than or equal to the value `x`.
Description:
This method searches the `set` for the first value which is greater than or equal to the given parameter `x` (it does not compare less than). Because sets are ordered containers, it means that all elements between the returned iterator and `set::end` will be greater than or equal to the searched value.
Name: `upper_bound`
Signature:
`iterator upper_bound ( const key_type& x ) const;`
Parameters:
`x`: the value to be looked for.
Return Value: The iterator to the first element which is greater than value `x`.
Description:
This method searches the `set` for the first value which is greater than (strict comparison) the given parameter `x`. Because sets are ordered containers, it means that all elements between the returned iterator and the `set::end` will be greater than the searched value.
Name: `equal_range`
Signature:
`pair<iterator,iterator> equal_range ( const key_type& x ) const;`
Parameters:
`x`: the value to be searched for in the `set/multiset`.
Return Value: A pair of iterators whose values depend on the result of the search:
if the search is successful, the first iterator points to the first element which is not less than (greater than or equal to) the requested value `x`, whereas the second iterator points to the first element greater than the given value `x`
if an element that is not less than `x` cannot be found, then both returned iterators point to the first element greater than `x`, or, if there is no greater element, to the past-the-end element
if an element that is not less than `x` is found, but an element greater than `x` is not, then the first returned iterator points to the found element, whereas the second iterator points to the past-the-end element.

Description:

The method `equal_range()` searches the `set` for the first element which is greater than or equal to value `x`, and the first element which is greater than value `x`.

Since the `set` is an ordered container, in the event of it being successful, the search produces a `range` of elements, containing exactly one element.

For a `multiset` container, the returned range can have more than one element. This function is just a combination of the `lower_bound()` and `upper_bound()` functions.

The first field of the returned pair of iterators should point to the element greater than or equal to `x` (`lower_bound()`), whereas the second field should point to an element greater than `x` (`upper_bound()`).

File name: 2.2.14.cpp

```
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template <class T> void print(T start, T end) {
        for ( ;start != end; ++start) {
                std::cout << *start << " ";
        }
    }

int main()
{
    int t[]={1, 10, 8, 4, 5, 6, 3, 9, 7, 2};
    set <int> s1(t,t+10);
    cout<<"S1: ";     print(s1.begin(), s1.end());cout<<endl;
    cout<<"Finding range [4,6]:\n";
    set<int>::iterator it1 = s1.lower_bound(4);
    set<int>::iterator it2 = s1.upper_bound(6);
    print(it1,it2); cout<<endl<<endl;
    cout<<"Finding single value range using equal_bounds\n";
    pair<set<int>::iterator, set<int>::iterator> p = s1.equal_range(4);
    print(p.first, p.second); cout<<endl;
    return 0;
}
```

References

Appendix A

How to install LaTeX

Windows OS

TeXLive package - full version

1. Download the TeXLive ISO (2.2GB) from
   https://www.tug.org/texlive/
2. Download WinCDEmu (if you don't have a virtual drive) from
   http://wincdemu.sysprogs.org/download/
3. To install Windows CD Emulator follow the instructions at
   http://wincdemu.sysprogs.org/tutorials/install/
4. Right click the iso and mount it using the WinCDEmu as shown in
   http://wincdemu.sysprogs.org/tutorials/mount/
5. Open your virtual drive and run setup.pl

or

Basic MikTeX - TeX distribution

1. Download Basic-MiKTeX(32bit or 64bit) from
   http://miktex.org/download
2. Run the installer
3. To add a new package go to Start » All Programs » MikTex » Maintenance (Admin) and choose Package Manager
4. Select or search for packages to install

TexStudio - TeX editor

1. Download TexStudio from
   http://texstudio.sourceforge.net/#downloads
2. Run the installer

Mac OS X

MacTeX - TeX distribution

1. Download the file from
   https://www.tug.org/mactex/
2. Extract and double click to run the installer. It does the entire configuration, sit back and relax.

TexStudio - TeX editor

1. Download TexStudio from
   http://texstudio.sourceforge.net/#downloads
2. Extract and Start

Unix/Linux

TeXLive - TeX distribution

Getting the distribution:

1. TexLive can be downloaded from
   http://www.tug.org/texlive/acquire-netinstall.html.
2. TexLive is provided by most operating system you can use (rpm,apt-get or yum) to get TexLive distributions

Installation

1. Mount the ISO file in the mnt directory

   ```
   mount -t iso9660 -o ro,loop,noauto /your/texlive####.iso /mnt
   ```

2. Install wget on your OS (use rpm, apt-get or yum install)
3. Run the installer script install-tl.

   ```
   cd /your/download/directory
   ./install-tl
   ```

4. Enter command ‘i’ for installation

5. Post-Installation configuration:
   http://www.tug.org/texlive/doc/texlive-en/texlive-en.html#x1-320003.4.1

6. Set the path for the directory of TexLive binaries in your .bashrc file

For 32bit OS

For Bourne-compatible shells such as bash, and using Intel x86 GNU/Linux and a default directory setup as an example, the file to edit might be

```
edit $~/.bashrc file and add following lines
PATH=/usr/local/texlive/2011/bin/i386-linux:$PATH;
export PATH
MANPATH=/usr/local/texlive/2011/texmf/doc/man:$MANPATH;
export MANPATH
INFOPATH=/usr/local/texlive/2011/texmf/doc/info:$INFOPATH;
export INFOPATH
```

For 64bit OS

```
edit $~/.bashrc file and add following lines
PATH=/usr/local/texlive/2011/bin/x86_64-linux:$PATH;
export PATH
MANPATH=/usr/local/texlive/2011/texmf/doc/man:$MANPATH;
export MANPATH
INFOPATH=/usr/local/texlive/2011/texmf/doc/info:$INFOPATH;
export INFOPATH
```

Fedora/RedHat/CentOS:

```
sudo yum install texlive
sudo yum install psutils
```

SUSE:

```
sudo zypper install texlive
```

Debian/Ubuntu:

```
sudo apt-get install texlive texlive-latex-extra
sudo apt-get install psutils
```

Appendix B

Installing the CUED class file

LaTeX.cls files can be accessed system-wide when they are placed in the <texmf>/tex/latex directory, where <texmf> is the root directory of the user's TeXinstallation. On systems that have a local texmf tree (<texmflocal>), which may be named "texmf-local" or "localtexmf", it may be advisable to install packages in <texmflocal>, rather than <texmf> as the contents of the former, unlike that of the latter, are preserved after the LaTeXsystem is reinstalled and/or upgraded.

It is recommended that the user create a subdirectory <texmf>/tex/latex/CUED for all CUED related LaTeXclass and package files. On some LaTeXsystems, the directory look-up tables will need to be refreshed after making additions or deletions to the system files. For TeXLive systems this is accomplished via executing "texhash" as root. MIKTeXusers can run "initexmf -u" to accomplish the same thing.

Users not willing or able to install the files system-wide can install them in their personal directories, but will then have to provide the path (full or relative) in addition to the filename when referring to them in LaTeX.