The Standard Template Library Tutorial

184.437 Wahlfachpraktikum (10.0)

Johannes Weidl

Information Systems Institute
Distributed Systems Department
Technical University Vienna

Advisor	Dipl. Ing. Georg Trausmuth
Professor	DI Dr. Mehdi Jazayeri

Friday, 26. April 1996

"The Standard Template Library (STL) is a C++ programming library that has been developed by Alexander Stepanov and Meng Lee at the Hewlett Packard laboratories in Palo Alto, California. It was designed to enable a C++ programmer to do generic programming and is based on the extensive use of templates - also called parametrized types. This paper tries to give a comprehensive and complete survey on the STL programming paradigm and shall serve as step-by-step tutorial for the STL newcomer, who has fundamental knowledge in C++ and the object-oriented paradigm."

Table of contents

1 Introduction

Motivation. In the late 70s Alexander Stepanov first observed that some algorithms do not depend on some particular implementation of a data structure but only on a few fundamental semantic properties of the structure. Such properties can be - for example - the ability, to get from one element of the data structure to the next, and to be able to step through the elements from the beginning to the end of the structure. For a sort algorithm it is not essential if the elements to be sorted are stored in an array, a linked list, etc. Stepanov examined a number of algorithms and found that most of them could be abstracted away from a particular implementation and that this abstraction can be done in a way that efficiency is not lost. Efficiency is an essential point that Stepanov emphasizes on, he is convinced that no one would use an algorithm that becomes inefficient by instantiating it back.

The STL history. Stepanovs insight - which hasn't had much influence on software development so far - will lead to a new programming paradigm in future - so the hope of its discoverer. In 1985 Stepanov developed a generic Ada library and was asked, if he could do this in C++ as well. But in 1987 templates (see section 2.3) - an essential technique for this style of programming - weren't implemented in C++ and so his work was delayed. In 1988 Stepanov moved to the HP Labs and 1992 he was appointed as manager of an algorithm project. Within this project, Alexander Stepanov and Meng Lee wrote a huge library - the Standard Template Library (STL) - with the intention to show that one can have algorithms defined as generically as possible without losing efficiency.

STL and the ANSI/ISO C++ Draft Standard. The importance of STL is not only founded in its creation or existence, STL was adopted into the draft standard at the July 14, 1994 ANSI/ISO C++ Standards Committee meeting. That means that if not happened till now anyway, compiler vendors will soon be incorporating STL into their products. The broad availability of STL and the generic programming idea give this new programming paradigm the chance to positively influence software development - thus allow programmers to write code faster and to write less lines of code while focusing more on problem solution instead of writing low-level algorithms and data structures.

Document arrangement. In section 2 STL-required C++ basics are taught, especially classes, function object design and templates - also called parametrized types. In section 3 STL is overviewed and the key concepts are explained. Section 4 teaches STL step-by-step. Section 5 deals with STL components not explained in section 4. Section 6 contains copyright notices and section 7 shows the literature used.

2 C++ basics

STL specific C++ basics. This section gives a short survey on STL-required C++ basics, such as classes, function objects and templates. It tries to point out the STL-specific aspects. For a fundamental and comprehensive study and understanding of these topics read [1], §5 to §8.

2.1 Classes

User-defined types. One reason to develop C into C++ was to enable and encourage the programmer to use the object-oriented paradigm. "The aim of the C++ class concept [...] is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types", says Bjarne Stroustrup, the father of C++, in [1]. It is stated that a class is a user-defined type:

```
class shape {
private:
   int x_pos;
   int y_pos;
   int color;
public:
   shape () : x_pos(0), y_pos(0), color(1) {}
   shape (int x, int y, int c = 1) : x_pos(x), y_pos(y), color(c) {}
   shape (const shape& s) : x_pos(s.x_pos), y_pos(s.y_pos), color(s.color) {}
  ~shape () {}
   shape& operator= (const shape& s) {
      x_pos = s.x_pos, y_pos = s.y_pos, color = s.color; return *this; }

   int get_x_pos () { return x_pos; }
   int get_y_pos () { return y_pos; }
   int get_color () { return color; }

   void set_x_pos (int x) { x_pos = x; }
   void set_y_pos (int y) { y_pos = y; }
   void set_color (int c) { color = c; }

   virtual void DrawShape () {}

   friend ostream& operator<< (ostream& os, const shape& s);
};

ostream& operator<< (ostream& os, const shape& s) {
      os <<  "shape: (" << s.x_pos << "," << s.y_pos << "," << s.color << ")";
      return os;
}
```

Examining the C++ class "shape". The keyword `class` begins the definition of the user-defined type. The keyword `private` means that the names `x_pos, y_pos` and `color` can only be used by member functions (which are functions defined inside the class definition). The keyword `public` starts the public-section, which constitutes the interface to objects of the class, that means, names and member functions in this section can be accessed by the user of the object. Because of the attributes being private, the class has public member functions to get and set the appropriate values. These member functions belong to the interface.
Note that a class is abstract, whereas the instantiation of a class leads to an object, which can be used and modified:

```
shape MyShape (12, 10, 4);

int color = MyShape.get_color();
shape NewShape = MyShape;
```

where `shape` is the class name and `MyClass` is an object of the class `shape`.

```
shape () : x_pos(0), y_pos(0), color(1) {}
```

is the default constructor - the constructor without arguments. A constructor builds and initializes an object, and there are more possible kinds of constructors:

```
shape (int x, int y, int c = 1) : x_pos(x), y_pos(y), color(c) {}
```

This is a constructor with three arguments where the third one is a default argument:

```
shape MyShape (10, 10);
```

results in: `x_pos == 10`, `y_pos == 10`, `color == 1`.

```
shape (const shape& s) : x_pos(s.x_pos), y_pos(s.y_pos), color(s.color) {}
```

This is an important constructor, the so-called copy-constructor. It is called when you write code like this:

```
shape MyShape;
shape NewShape (MyShape);
```

After that, `MyShape` and `NewShape` have the same attributes, the object `NewShape` is copied from the object `MyShape` using the copy constructor.
Note the argument `const shape& s`. The & means "reference to", when a function call takes place, the shape is not copied onto the stack, but only a reference (pointer) to it. This is important, when the object given as argument is huge, because then copying would be very inefficient.

```
~shape () {}
```

is the destructor. It is called, when an object is destroyed - for example when it goes out of scope. The `shape` destructor has nothing to do, because inside the shape class no dynamically allocated memory is used.

```
shape& operator= (const shape& s) {
      x_pos = s.x_pos, y_pos = s.y_pos, color = s.color; return *this; }
```

Operator overloading. In C++ it is possible to *overload* operators - that is to give them a new meaning or functionality. There is a set of operators which can be defined as member functions inside a class. Among these the assignment operator can be found, which is used when writing the following code:

```
shape MyShape, NewShape;
NewShape = MyShape;
```

Note that the `operator=` is called for the left object, i.e. `NewShape`, so there must be only one argument in the declaration. This is true for all other C++ operators as well.
When a member function is called, the system automatically adds the `this`-pointer to the argument list. The `this`-pointer points to the object, for which the member function is called. By writing `return *this`, the concatenation of assignments gets possible:

```
shape OldShape, MyShape, NewShape;
NewShape = MyShape = OldShape;

int get_x_pos () { return x_pos; }
```

gives you the value of `x_pos`. An explicit interface function is necessary, because private mebers cannot be accessed from outside the object.

```
virtual void DrawShape () {}
```

declarates a function with no arguments that draws the shape. Because a shape is abstract and we have no idea of what it looks like precisely, there's no implementation for `DrawShape`. The keyword `virtual` means that this member function can be overwritten in a *derived* class (see [1], §6). For example, a class `dot` could be derived from `shape`. `DrawShape` then would be overwritten to draw the dot at the position (`x_pos`,`y_pos`) and with the colour `color`.

Put-to operator. Now consider the definition of the `operator<<`:

```
ostream& operator<< (ostream& os, const shape& s) {
	os <<  "shape: (" << s.x_pos << "," << s.y_pos << "," << s.color << ")";
	return os;
}
```

The usual way in C++ to display information on the screen is to write:

```
cout << "Hello, World!";
```

With the upper code we overload the *put-to-operator* (`operator<<`) to be able to send shapes directly to an output stream:

```
shape MyShape (5, 9);
cout << MyShape;
```

shows on the output screen: *shape: (5,9,1)*

```
friend ostream& operator<< (ostream& os, const shape& s);
```

Friend and inline. The keyword `friend` in front of a function declaration means that this function has access to the private members of the class, where the declaration takes place. You can see that `x_pos`, `y_pos` and `color` are used directly by `operator<<`. It's also possible to define a whole class as *friend* class.
Note that all member functions of `shape` are defined inside the class declaration. If so, the member functions are all "*inline*". *Inline* means, that wherever the function is called, the compiler creates no function call but inserts the code directly to decrease overhead.
To inline a member function defined outside the class the keyword `inline` must be used:

```
inline int shape::get_x_pos () { return x_pos; }
```

Nice Classes. For STL it's wise to create classes that meet the requirements of *Nice Classes*. For example, Borland C++ expects an object to be stored in a container to have an assignment operator defined. Additionally, if a container holds its objects in a particular order, a operator like the `operator<` must be defined (the latter to fix a half-order).

A class T is called *nice* iff it supports:

1. Copy constructor `T (const T&)`
2. Assignment operator `T& operator= (const T&)`
3. Equality operator `int operator== (const T&, const T&)`
4. Inequality operator `int operator!= (const T&, const t&)`

such that:

1. `T a(b); assert (a == b);`
2. `a = b; assert (a == b);`
3. `a == a;`
4. `a == b iff b == a`
5. `(a == b) && (b == c) implies (a == c)`
6. `a != b iff ! (a == b)`

A member function `T::s(...)` is called *equality preserving* iff

```
a == b implies a.s (...) == b.s (...)
```

A class is called Extra-Nice iff
all of its member functions are equality preserving

The theory of Nice Classes origins from a joint work between HP and Andrew Koenig from the Bell Labs.

2.2 Function objects

The function-call operator. A function object is an object that has the *function-call operator* (`operator()`) defined (or overloaded).
These function objects are of crucial importance when using STL.
Consider an example:

```
class less {
public:
   less (int v) : val (v) {}
   int operator () (int v) {
      return v < val;
   }
private:
   int val;
};
```

This function object must be created by specifying an integer value:

```
less   less_than_five (5);
```

The constructor is called and the value of the argument `v` is assigned to the private member `val`. When the function object is applied, the return value of the overloaded function call operator tells if the argument passed to the function object is less than `val`:

```
cout << "2 is less than 5: " << (less_than_five (2) ? "yes" : "no");
```

Output: *2 is less than 5: yes*

You should get familiar with this kind of programming, because when using STL you often have to pass such function objects as arguments to algorithms and as template arguments when instantiating containers, respectively.

2.3 Templates

Static type checking. C++ is a language that supports static type checking. Static type checking helps to catch many errors during compilation, because the programmer has to fix the type of a name used. Any violation of the type model leads to an error message and cancels compilation. So, run-time errors decrease.

2.3.1 Function templates

Consider the following function:

```
void swap (int& a, int& b) {

   int tmp = a;
   a = b;
   b = tmp;
}
```

Swapping integers. This function let's you swap the contents of two integer variables. But when programming quite a big application, it is probable that you have to swap float, long or char variables, or even `shape` variables - as defined in section 2. So, an obvious thing to do would be to copy the piece of code (cut-n-paste!) and to replace all `int`s by `shape`s, wouldn't it?
A drawback of this solution is the number of similar code pieces, that have to be administered. Additionally, when you need a new swap function, you must not forget to code it, otherwise you get a compile-time error. And now imagine the overhead when you decide to change the return type from `void` to `int` to get information, if the swap was successful - the memory could be too low to create the local `tmp` variable, or the assignment operator (see `shape`) could not be defined. You would have to change all x versions of `swap` - and go insane...

Templates or Parametrized types. The solution to this dark-drawn scenario are templates, template functions are functions that are parametrized by at least one type of their arguments:

```
template <class T>
void swap (T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

Note that the ″T″ is an arbitrary type-name, you could use ″U″ or ″anyType″ as well. The arguments are references to the objects, so the objects are not copied to the stack when the function is called. When you write code like

```
int a = 3, b = 5;
shape MyShape, YourShape;

swap (a, b);
swap (MyShape, YourShape);
```

the compiler "instantiates" the needed versions of `swap`, that means, the appropriate code is generated. There are different template instantiation techniques, for example manual instantiation, where the programmer himself tells the compiler, for wich types the template should be instantiated.

Function template examples. Other examples for function templates are:

```
template <class T>
T& min (T& a, T&b) { return a < b ? a : b; }


template <class T>
void print_to_cout (char* msg, T& obj) {
   cout << msg << ": " << obj << endl;
}
```

To use the last template function, objects given as the second argument must have the `operator<<` defined, otherwise you will get a compile-time error.

2.3.2 Class templates

Class templates to build containers. The motivation to create class templates is closely related to the use of containers. "However, container classes have the interesting property that the type of objects they contain is of little interest to the definer of a container class, but of crucial importance to the user of the particular container. Thus we want to have the type of the contained object be an argument to a container class: [...]", [1], §8. That means that a container - e.g. a vector - should be able to contain objects of any type. This is achieved by class templates. The following example comes from [1], §1.4.3:

```
template <class T>
class vector {
   T* v;
   int sz;
public:
   vector (int s) { v = new T [sz = s]; }
  ~vector () { delete[] v; }
   T& operator[] (int i) { return v[i]; }
   int get_size() { return sz; }
};
```

Note that no error-checking is done in this example. You can instantiate different vector-containers which store objects of different types:

```
vector<int>       int_vector (10);
vector<char>      char_vector (10);
vector<shape>     shape_vector (10);
```

Take a look at the notation, the type-name is *vector<specific_type>*.

2.3.3 Template member functions

By now there’s no compiler I know which could handle template member functions. This will change in the very future, because template member functions are designated in the C++ standard.

2.3.4 Template specialization

Cope with special type features. If there is a good reason, why a compiler-generated template for a special type does not meet your requirements or would be more efficient or convenient to use when implemented in another way, you can give the compiler a special implementation for this type - this special implementation is called *template specialization*. For example, when you know, that a `shape`-vector will always hold exactly one object, you can specialize the `vector`-template as follows:

```
class vector<shape> {
   shape v;
public:
   vector (shape& s) : v(s) { }
   shape& operator[] (int i) { return v; }
   int get_size() { return 1; }
};
```

Let’s use it:

```
shape              MyShape;
vector<shape>      single_shape_vector (MyShape);
```

Template specializations can also be provided for template functions ([1], §r.14.5) and template operators.

3 A STL overview

STL is a component library. This means that it consists of components - clean and formally sound concepts. Such components are for example containers - that are objects which store objects of an arbitrary type - and algorithms. Because of the generic approach STL algorithms are able to work on user-built containers and user-built algorithms can work on STL containers - if the user takes some strict requirements for building his components into consideration. This technique - to guarantee the interoperability between all built-in and user-built components - is referred to as "the orthogonal decomposition of the component space". The idea behind STL can easily be shown by the following consideration:

Imagine software components as a three-dimensional space. One dimension represents the data types (int, double, char, ...), the second dimension represents the containers (array, linked-list, ...) and the third dimension represents the algorithms (sort, merge, search, ...).

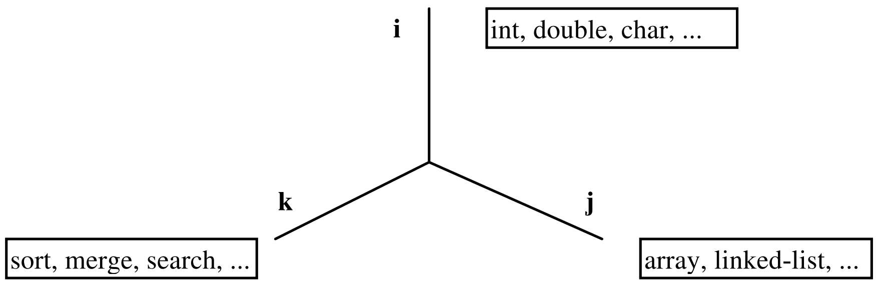


Figure 1: Component space

With this scenario given, i*j*k different versions of code have to be designed - a sort algorithm for an array of int, the same sort algorithm for an array of double, a search algorithm for a linked-list of double and so on. By using template functions that are parametrized by a data type, the i-axes can be dropped and only j*k versions of code have to be designed, because there has to be only one linked-list implementation which then can hold objects of any data-type. The next step is to make the algorithms work on different containers - that means that a search algorithm should work on arrays as well as on linked-lists, etc. Then, only j+k versions of code have to be created.

STL embodies the above concept and is thus expected to simplify software development by decreasing development times, simplifying debugging and maintenance and increasing the portability of code.

STL consists of five main components. When I list them here, don't get confused by the names and their short description, they are explained one by one in detail later.

- Algorithm: computational procedure that is able to work on different containers
- Container: object that is able to keep and administer objects
- Iterator: abstraction of the algorithm-access to containers so that an algorithm is able to work on different containers
- Function Object: a class that has the function-call operator (`operator()`) defined
- Adaptor: encapsulates a component to provide another interface (e.g. make a stack out of a list)

At this point I recommend to read [2], chapters 1 to 4.

3.1 STL availability and information

3.1.1 FTP-Sites

The Hewlett Packard STL by Alexander Stepanov and Meng Lee can be found at:

ftp://butler.hpl.hp.com/pub/stl/stl.zip for Borland C++ 4.x
ftp://butler.hpl.hp.com/pub/stl/sharfile.Z for GCC

There are many other interesting things there, too. An alternative site is

ftp://ftp.cs.rpi.edu/stl

This document deals with the HP implementation of STL, but there are others to:

ObjectSpace STL<ToolKit>

FSF/GNU libg++ 2.6.2:
ftp://prep.ai.mit.edu/pub/gnu/libg++-2.6.2.tar.gz

Both work with the GNU C++ compiler GCC 2.6.3 that can be found at:
ftp://prep.ai.mit.edu/pub/gnu/gcc-2.6.3.tar.gz

Especially for the work with ObjectSpace STL<ToolKit> you should patch your GCC 2.6.3 with the template fix that can be found at
ftp://ftp.cygnus.com/pub/g++/gcc-2.6.3-template-fix

Many examples for the ObjectSpace STL<ToolKit> can be found at
ftp://butler.hpl.hp.com/pub/stl/examples.gz (also .zip)

3.1.2 URLs

David Mussers STL-page:
http://www.cs.rpi.edu/~musser/stl.html
Mumit's STL Newbie guide:
http://www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html
Joseph Y. Laurino's STL page:
http://weber.u.washington.edu/~bytewave/bytewave_stl.html

3.2 What does STL consist of?

Here comes a list of the files included in the HP-STL .ZIP package with the HASH extension:

DOC.PS	STL Document [2]
DOCBAR.PS	STL Document [2] with changebars from the previous version
IMP.PS	
FILES.DIF	Differences to the files of the previous version
READ.ME	Information file
README.OLD	Information file of the previous version
ALGO.H	algorithm implementations
ALGOBASE.H	auxiliary algorithms for ALGO.H
ITERATOR.H	iterator implementations and iterator adaptors
FUNCTION.H	operators, functions objects and function adaptors
TREE.H	implementation of a red-black tree for associative containers
BOOL.H	defines bool type
PAIR.H	defines pair type to hold two objects
TRIPLE.H	defines triple type to hold three objects
HEAP.H	heap algorithms
STACK.H	includes all container adaptors
HASH.H	hash implementation
HASHBASE.H	hashbase implementation needed by hash
TEMPBUF.CPP	auxiliary buffer for get_temporary_buffer: should be complied and linked if get_temporary_buffer, stable_partition, inplace_merge or stable_sort are used
TEMPBUF.H	get_temporary_buffer implementation
PROJECTN.H	select1st and ident implementation
RANDOM.CPP	random number generator, should be compiled and linked if random_shuffle is used
DEFALLOC.H	default allocator to encapsulate memory model
BVECTOR.H	bit vector (vector template specialization), sequence container
DEQUE.H	double ended queue, seuqence container
LIST.H	list, sequence container
MAP.H	map, associative container
MULTIMAP.H	multimap, associative container
SET.H	set, associative container
MULTISET.H	multiset, associative container
VECTOR.H	vector, sequence container

Dos/Windows specific include files:

Huge memory model:
HUGALLOC.H, HDEQUE.H, HLIST.H, HMAP.H, HMULTMAP.H, HMULTSET.H, HSET.H, HVECTOR.H

Far memory model:
FARALLOC.H, FDEQUE.H, FLIST.H, FMAP.H, FMULTMAP.H, FMULTSET.H, FSET.H

Large memory model:
LNGALLOC.H, LBVECTOR.H, LDEQUE.H, LLIST.H, LMAP.H, LMULTMAP.H, LMULTSET.H, LSET.H

Near memory model:
NERALLOC.H, NMAP.H, NMULTMAP.H, NMULTSET.H, NSET.H

Table 1: STL include and documentation files

3.3 Compiling STL programs

3.3.1 Borland C++ 4.0 DOS-programs

Command Line.

Assume a C++ program named `vector.cpp`:

```
#define __MINMAX_DEFINED // use STL's generic min and max templates
#define __USE_STL        // exclude BC++'s redundant operator definitions

// STL include files - include STL files first!
#include "vector.h"

// C++ standard include files
#include <stdlib.h>             // stdlib min and max functions are skipped
#include <cstring.h>            // only compilable with __USE_STL directive
#include <classlib\alloctr.h>   // only compilable with __USE_STL directive
#include <iostream.h>

void main (void)
{
  vector<int> v(5);
  v[0] = 4;
  cout << "First vector element: " << v[0];
}
```

The compiler directive `#define __MINMAX_DEFINED` prevents the compilation of the `min` and `max` functions in the Borland C++ include file `<stdlib.h>`, because STL provides its own template `min` and `max` functions.
I recommend to include <u>all</u> STL include files before the Borland C++ standard include files, although this causes some work to be done.
There are some changes to be made in the include files `<bc4\include\cstring.h>` and `<bc4\include\classlib\alloctr.h>`, if you plan to use them. Some operator definitions have to be taken out of compilation, for example by adding

```
#if !defined (__USE_STL) [...] #endif,
```

because STL generates these operators automatically using template operator definitions.

The code after adding the necessary `#if` directives (italic letters) is shown in the following box. The line numbers indicate the operator-definition-positions in the original include files:

```
<bc4\include\cstring.h>:
line 724:
#if !defined(__USE_STL)
inline int _RTLENTRY operator != ( const string _FAR &s1, const string
_FAR &s2 )  THROW_NONE
{ [...] }
#endif

line 850:
#if !defined(__USE_STL)
inline int _RTLENTRY operator <= ( const string _FAR &s1, const string
_FAR &s2 )  THROW_NONE
{ [...] }
#endif

line 866:
#if !defined(__USE_STL)
inline int _RTLENTRY operator >  ( const string _FAR &s1, const string
_FAR &s2 )  THROW_NONE
{ [...] }
#endif
```

```
line 882:
#if !defined(__USE_STL)
inline  _RTLENTRY operator >= ( const string _FAR &s1, const string _FAR
&s2 )THROW_NONE
{ [...] }
#endif

<bc4\include\classlib\alloctr.h>, line 44:

#if !defined(__USE_STL)
    friend void *operator new( unsigned, void *ptr )
        { return ptr; }
#endif
```

Compile and link `.cpp` files using STL with the following command:

```
bcc -I<path-to-stl-directory> <file>.cpp
```

Example:

```
bcc -Ic:\bc4\stl vector.cpp
```

It is also possible to include the STL include files after the Borland C++ standard include files, then programs would even compile without having changes in `<bc4\include\cstring.h>`. But STL provides a number of template functions that increase genericity and template operator definitions that generate `operator!=` out of `operator==` and operators `>`, `>=`, `<=` out of `operator<`, so it seems advisable to choose the practice shown above.

IDE (Integrated Development Environment).

Create a project specifying "DOS-Standard" as target-platform. Specify the STL-directory under "options/project/directories" (german: "Optionen/Projekt/Verzeichnisse") as include-directory. Use the `#define __MINMAX_DEFINED` statement when `<stdlib.h>` is included, use `#define __USE_STL` when `<cstring.h>` and `<classlib\alloctr.h>` are included.

3.3.2 Borland C++ 4.0 WINDOWS-programs

As under DOS, the `#define __MINMAX_DEFINED` statement is needed when `<stdlib.h>` is included. Use `#define __USE_STL` to compile your programs, when using `<cstring.h>` and `<classlib\alloctr.h>`. Don't forget to specify the STL-directory as include-directory under "options/project/directories" (german: "Optionen/Projekt/Verzeichnisse").

Example program:

```
#define __MINMAX_DEFINED // use STL's generic min and max templates
#define __USE_STL        // exclude BC++'s redundant operator definitions

// STL include files
#include "vector.h"
#include "algo.h"

// C++ standard include files
#include <stdlib.h>             // stdlib min and max functions are skipped
#include <cstring.h>            // only compilable with __USE_STL directive
#include <classlib\alloctr.h>   // only compilable with __USE_STL directive

// OWL2 include files
```

```
#include <owl\owlpch.h>
#include <owl\applicat.h>

int OwlMain(int /*argc*/, char* /*argv*/ [])
{
  return TApplication("Compiled with STL include files").Run();
}
```

I encountered some problems when compiling windows programs that make extensive use of STL containers. The compiler comes up with the error messages `"code segment exceeds 64k"` and `"text segment exceeds 64k"`. The problem can be fixed by using the statements `#pragma codeseg <codeseg_name> code` and `#pragma codeseg <textseg_name> text`, respectively.

3.3.3 Borland C++ 4.5 DOS- and WINDOWS-programs

For programs written in Borland C++ 4.5 all information given in sections 3.3.1 and 3.3.2 can be applied but there are some further points:

- The first include file has to be `<classlib\defs.h>`, because there Borland C++ defines its `bool` type.
- Then, all STL include files have to be included (before any Borland C++ include files).
- Note, that the line numbers of operators that have to be commented out by a `#define __USE_STL` directive in the include files `<cstring.h>` and `<classlib\alloctr.h>` are not the same as given in section 3.3.1 for the appropriate Borland C++ 4.0 include files.
- A further operator has to be excluded by a `#define __USE_STL` directive in the include file `<owl\bitset.h>` (found at the end of the include file), if it is used.

DOS-example (analogous for Windows):

```
#define __MINMAX_DEFINED // use STL's generic min and max templates
#define __USE_STL        // exclude BC++'s redundant operator definitions

#include <classlib\defs.h>     // use BC++4.5 bool definition

// STL include files
#include "vector.h"

// C++ standard include files
#include <stdlib.h>
#include <cstring.h>
#include <classlib\alloctr.h>
#include <owl\bitset.h>
#include <iostream.h>

void main (void)
{
  vector<int> v(1, 4);
  cout << v[0];
}
```

4 Learning STL

4.1 Containers

As Bjarne Stroustrup says, "One of the most useful kinds of classes is the container class, that is, a class that holds objects of some (other) type", [1], §8.1. Containers form one crucial component of STL. To sum up elements of a special type in a data structure, e.g. temperature values of an engine over a definite distance of time, is a crucial task when writing any kind of software. Containers differ in the way how the elements are arranged and if they are sorted using some kind of key.

In STL you find *Sequence Containers* and *Associative Containers*. As described in [2], "A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement". STL provides three basic kinds of Sequence Containers: *Vector*s, *List*s and *Deque*s, where Deque is an abbreviation for *Double Ended Queue*.

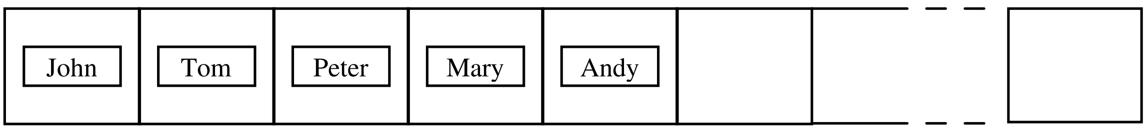


Figure 2: Sequence Container

As Stepanov states, "Associative containers provide an ability for fast retrieval of data based on keys". The elements are sorted and so fast binary search is possible for data retrieval. STL provides four basic kinds of Associative Containers. If the key value must be unique in the container, this means, if for each key value only one element can be stored, *Set* and *Map* can be used. If more than one element are to be stored using the same key, *Multiset* and *Multimap* are provided.

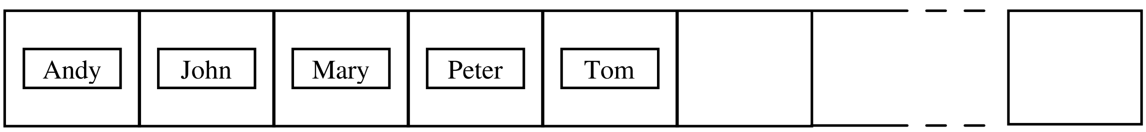


Figure 3: Associative Container

Here is a summary including all containers provided by STL:

Sequence Containers	**Vector**
	Deque
	List
Associative Containers	**Set**
	Multiset
	Map
	Multimap

Table 2: STL Containers

4.1.1 Vector

Assume we want to develop a Graphical User Interface for a control station in an electric power station. The single elements, like turbines, pipes and electrical installations are shown on a screen. For each power station element we derive a special class from the `shape` class in section 2 to represent its look on the screen. The class hierarchy could look like this:

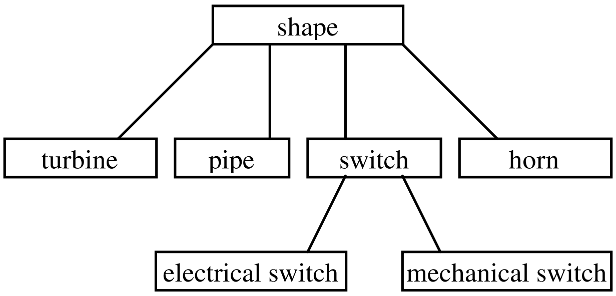


Figure 4: Example shape class hierarchy

We store all shapes that are shown on a certain screen in the appropriate shape-container, e.g. all turbine objects that are shown on the main screen in a turbine-container. When the screen is called, the containers are used to draw a representation of the appropriate part of the power station.

In C++ one could use an array:

```
turbine main_screen_turbines [max_size];
```

where `max_size` is the maximum number of turbine objects that can be stored in the `main_screen_turbines` array.

When you use STL, you would choose this:

```
#include <vector.h>

typdef int turbine;      // so we don´t have to define the turbine class

int main() {

   vector<turbine> main_screen_turbines;
   return 0;
}
```

Note: To make this little example run you have to read section 3.3 on how to compile STL programs. To use a vector in your program, include `vector.h`. In the following examples only the essential code lines are presented and most of the include stuff and the main function are omitted.

As you can see, you don't have to specify a maximum size for the vector, because the vector itself is able to dynamically expand its size. The maximum size the vector can reach - i.e. the maximum number of elements it is able to store - is returned by the member function `max_size()` of the `vector` class:

```
vector<turbine>::size_type max_size = main_screen_turbines.max_size();
```

Note: Many member functions described in the vector-section can be found among the rest of the STL containers, too. The description applies to those containers accordingly and will be referenced when discussing these containers.

`size_type` is an unsigned integral type, this could be for example `unsigned long`. The type that determines the size of the different containers is encapsulated by a `typedef` to abstract from the actual memory model used. For example:

```
typedef unsigned long size_type;
```

if the size is expressible by the built in type `unsigned long`.

STL abstracts from the specific memory model used by a concept named *allocators*. All the information about the memory model is encapsulated in the `Allocator` class. Each container is templatized (parametrized) by such an *allocator* to let the implementation be unchanged when switching memory models.

```
template <class T, template <class U> class Allocator = allocator>
class vector {
   ...
};
```

The second template argument is a default argument that uses the pre-defined allocator "`allocator`", when no other allocator is specified by the user. I will describe allocators in detail in section 4.5.

If you want to know the actual size of the vector - i.e. how many elements it stores at the moment - you have to use the `size()` member function:

```
vector<turbine> main_screen_turbines;

vector<turbine>::size_type size  = main_screen_turbines.size();
cout << "actual vector size: " << size;
```

Output: *actual vector size: 0*

Like `size_type` describes the type used to express the size of a container, `value_type` gives you the type of the objects that can be stored in it:

```
vector<float> v;
cout << "value type: " << typeid[1] (vector<float>::value_type).name();
```

Output: *value type: float*

A container turns out useless if no object can be inserted into or deleted from it. The vector, of course, provides member functions to do these jobs and it does quite a bit more:
It is *guaranteed* that inserting and erasing at the end of the vector takes *amortized* constant time whereas inserting and erasing in the middle takes linear time.
As stated in [3], R-5, "In several cases, the most useful characterization of an algorithm's computing time is neither worst case time nor average time, but *amortized time*. [...] Amortized time can be a useful way to describe the time taken by an operation on some

[1] To use `typeid` include `typeinfo.h`

container in cases *where the time can vary widely* as a sequence of the operations is done, but the total time for a sequence of N operations has a *better bound* than just N times the worst-case time." To understand this, remember that a vector is able to automatically expand its size. This expansion is done, when an insert command is issued but no room is left in the storage allocated. In that case, STL allocates room for $2n$ elements (where n is the actual size of the container) and copies the n existing elements into the new storage. This allocation and the copying process take linear time. Then the new element is inserted and for the next $n-1$ insertions only constant time is needed. So you need $O(n)$ time for n insertions, averaged over the n insert operations this results in $O(1)$ time for one insert operation. This more accurately reflects the cost of inserting than using the worst-case time $O(n)$ for each insert operation.

Of course amortized constant time is about the same overhead as you have when using C/C++ arrays but note that it is important to be about the same - and not *more*.
For the authors of STL complexity considerations are very important because they are convinced that component programming and especially STL will only be accepted when there is no (serious) loss of efficiency when using it.
Maybe there are users who can afford to work inefficiently but well designed - most can not.

The following table shows the insert and erase overheads of the containers `vector`, `list` and `deque`. Think of these overheads when choosing a container for solving a specific task.

Container	***insert/erase overhead at the beginning***	***in the middle***	***at the end***
Vector	linear	linear	amortized constant
List	constant	constant	constant
Deque	amortized constant	linear	amortized constant

Table 3: Insert and erase overheads for vector, list and deque

Before we look at the insert functionality, there is another thing to consider. When a vector is constructed using the default constructor (the default constructor is used when no argument is given at the declaration), no memory for elements is allocated:

```
vector<int> v;
```

We can check this using the member function `capacity()`, which shows the number of elements for which memory has been allocated:

```
vector<int>::size_type capacity = v.capacity();
cout << "capacity: " << capacity;
```

<u>Output:</u> *capacity: 0*

At the first glance this doesn't make any sense but it gets clear when you consider, that the vector class itself is able to allocate memory for the objects inserted. In C++ you would fill your turbine array as follows:

```
turbine turb;
turbine main_screen_turbines [max_size];

main_screen_turbines[0] = turb;
```

In STL you can use this syntax, too:

```
turbine turb;
vector<turbine> main_screen_turbines (10);  // allocate memory for 10
                                            // elements
main_screen_turbines[0] = turb;
```

Now, we don't use the default constructor but specify a number that tells the vector for how many elements memory should be allocated. Then we use the overloaded subscribe operator (`operator[]`) to insert a turbine object into the vector.

> **Note:** If you use the subscribe operator with an index, for which no memory has been allocated (this is true for all indices when declaring a vector without specifying a vector size), the result will be undefined!

To avoid memory allocation stuff the vector provides different member functions to insert elements into the vector. These insert functions do automatic memory allocation and - if necessary - expansion. To append an element at the end of a vector use `push_back()`:

```
vector<int> v;

v.push_back (3);
cout << v.capacity() << endl;
cout << v[0];
```

Output: *2048* [2]
3

Three different (overloaded) kinds of `insert()` member functions can be used. Here comes the first:

```
vector<int> v;

v.insert (v.end(), 3);
cout << v.capacity() << endl;
cout << v[0];
```

Output: *2048*
3

This first kind of the `insert()` member function needs two arguments: an *iterator* "pointing" to a definite container position and an element which is to be inserted.
The element is inserted before the specified *iterator-position*, that is before the element the specified iterator points to.
The term *iterator* needs some explanation. There are two member functions which return so-called iterators: `begin()` and `end()`.
Iterators are a generalization of the C++ pointers. An iterator is a kind of pointer but indeed *more* than a pointer. Like a pointer is dereferenced by the expression **pointer*, an iterator has the dereference `operator*` defined which returns a value of a specific type - the value type of the iterator. Additionally, like a pointer can be incremented by using the `operator++`, an iterator can be incremented in the same way. Iterators most often are associated with a container. In that case, the value type of the iterator is the value type of the container and dereferencing the iterator returns an object of this value type. Look at this example to get a feeling how iterators behave:

[2] This value depends on the environment (memory model) used

```
vector<int> v(3);

v[0] = 5;
v[1] = 2;
v[2] = 7;

vector<int>::iterator first = v.begin();
vector<int>::iterator last  = v.end();


while (first != last)

   cout << *first++ << " ";
```

<u>Output:</u> *5 2 7*

`v.begin()` returns an iterator to the first element in the vector. The iterator can be dereferenced and incremented like a C++ pointer.
Please note, that `v.end()` doesn't return an iterator that points to the last element in the vector - as now could be supposed - but *past* the last element (however, in the STL code such an iterator is named `last`). Accordingly it is called *past-the-end* iterator. A user is not supposed to dereference such an iterator, because the result would be undefined. The `while` loop checks if the `first` iterator is equal to the `last` iterator. If not, the iterator is dereferenced to get the object it is pointing to, then it is incremented. So, all vector elements are written to `cout`.

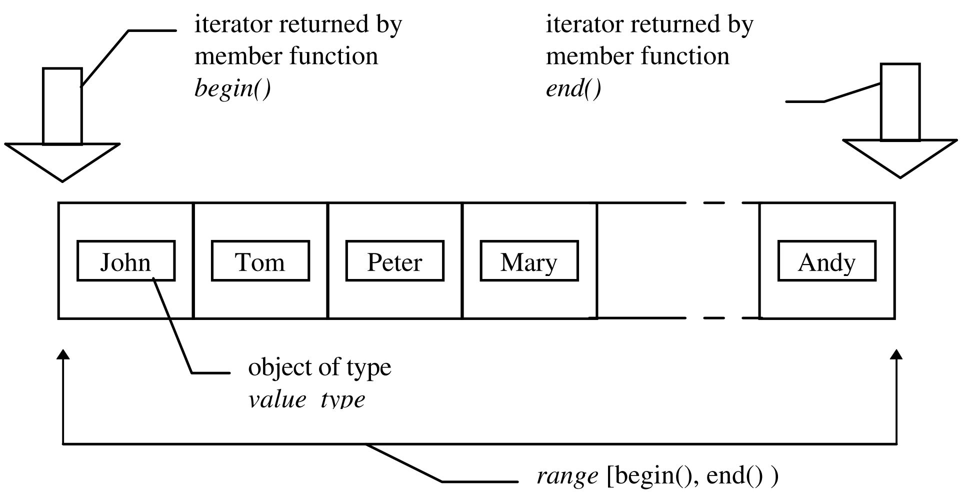


Figure 5: Range specified by iterators

A *range* `[i, j)` given by the iterators `i` and `j` is valid, if `j` is *reachable* from `i`, that means if there is a finite sequence of applications of `operator++` to `i` that makes `i==j`;
Ranges given by two iterators are very important in STL, because STL algorithms largely work in the following way:

```
sort (begin-iterator, past_the_end-iterator)
```

where `begin-iterator` specifies the first element in the range and `past_the_end-iterator` points past the last element of the range to be sorted.
The range is correctly specified by the expression `[begin-iterator, past_the_end-iterator)`.

A valid sort command for our vector-example would be:

```
sort3 (v.begin(), v.end() );
```

Using iterators as intermediates, we are able to seperate the algorithms from the container implementations:

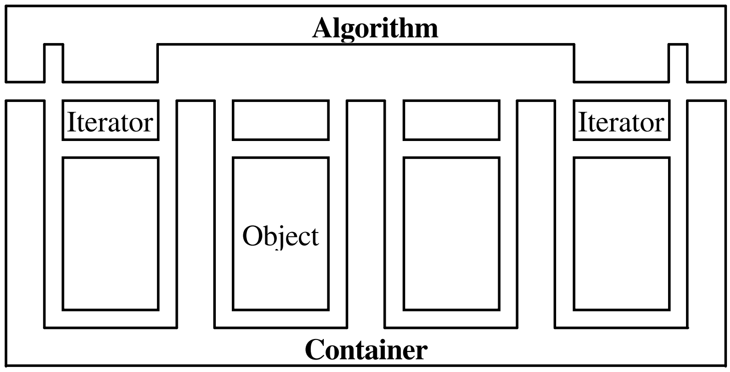


Figure 6: Orthogonal decomposition of the component space

After this short survey on iterators, which will be described in very detail in the next section, we focus on the vector container again.
We learned that specifying a number when declaring a vector reserves memory for elements. Additionally to that, you can give the elements for which memory is reserved an initial value:

```
vector<int> v(3, 17);
for (int i = 0; i < 3; i++) cout << v[i] << " ";
```

<u>Output:</u> *17 17 17*

It is possible to construct a vector out of another or to assign one vector to another vector:

```
vector<float> v (5, 3.25);

vector<float> v_new1 (v);                  // construct v_new1 out of v
vector<float> v_new2 = v;                  // assign v to vnew2
vector<float> v_new3 (v.begin(), v.end() );
                     // construct v_new3 out of the elements of v
```

The last version uses iterators to specify the range out of which the `v_new3` vector should be constructed. The three `v_new` - vectors are all equal:

```
(v_new1 == v_new2) && (v_new2 == v_new3) && (v_new1 == v_new3) ? \
cout << "equal" : cout << "different";
```

<u>Output:</u> *equal*

To be able to compare vectors, an equality `operator==` for vectors is provided.

To swap two vectors, a special member function is provided which needs merely constant time, because only internal pointers are manipulated.

[3] To use algorithms in your programs you have to include `algo.h`

```
vector<int> v (1, 10);
vector<int> w (1, 20);

v.swap (w);
cout << v[0];
```

Output: *20*

With the member function `empty()` one can test if a vector is empty, i.e. if its size is zero:

```
vector<char> v;
v.empty() ? cout << "empty" : cout << "not empty";
```

Output: *empty*

The first and the last element are returned when invoking `front()` and `back()`:

```
vector<int> v (10, 5);
v.push_back (7);
cout << v.front() << " " << v.back();
```

Output: *5 7*

With `pop_back()` the last element is returned and deleted from the vector.

```
vector<int> v (1, 2);
int value = v.pop_back ();
cout << value << endl;
v.empty() ? cout << "empty" : cout << "not empty";
```

Output: *2*
empty

Additionally to the `insert()` member function that takes an iterator and an element as arguments, two more versions are provided:

```
vector<int> v;
v.insert (v.begin(), 2, 5);                // vector v: 5 5

vector<int> w (1, 3);
w.insert (w.end(), v.begin(), v.end() );    // vector w: 3 5 5
```

The second argument of the first version specifies how many copies of an element - given as third argument - should be inserted before the specified iterator-position (first argument). The second version takes additionally to the inserting position `w.end()` two iterators that specify the range which is to be inserted.

Using the `erase()` member function, it is possible to erase single elements or ranges (specified by two iterators) from a vector. Accordingly, there are two versions of `erase()`. Erasing at the end of the vector takes constant time whereas erasing in the middle takes linear time.

```
vector<float> v (4, 8.0);       // vector v:  8.0  8.0  8.0  8.0
v.erase (v.begin() );           // vector v:  8.0  8.0  8.0
v.erase (v.begin(), v.end() ); // vector v:
```

The first version erases the first vector element. The second version erases all remaining elements so the vector gets empty.

When inserting in or erasing from a container, there is something to take into consideration. If you have an iterator pointing e.g. to the end of a vector and you insert an element at its beginning, the iterator to the end gets *invalid*. Only iterators before the insertion point remain valid. If no place is left and expansion takes place, all iterators get invalid. This is clear, because new memory is allocated, the elements are copied and the old memory is freed. Iterators aren't automatically updated and get invalid, that means the result of operations using such iterators is undefined. Take this into consideration when inserting or erasing and then using iterators earlier defined on this container. The following table shows the validity of the containers `vector`, `list` and `deque` after inserting and erasing an element, respectively.

Container	***operation***	***iterator validity***
vector	inserting	reallocation necessary - all iterators get invalid
		no reallocation - all iterators before insert point remain valid
	erasing	all iterators after erasee point get invalid
list	inserting	all iterators remain valid
	erasing	only iterators to erased elements get invalid
deque	inserting	all iterators get invalid
	erasing	all iterators get invalid

Table 4: Iterator validity after inserting or erasing

Now we are able to store objects in a container (at least in the vector) that provides several means to administer and maintain it. To apply algorithms to the elements in the vector we have to understand the iterator concept which is described in detail in the next section.

4.1.2 Exercises

This section contains specifications for exercises dealing with the topics in section 4.1. Solving these tasks should give you the possibility to apply your lections learned and compare your solutions with the ones given in the solutions part of this tutorial.

Exercise 4.1.1: Write a STL program that declares a vector of integer values, stores five arbitrary values in the vector and then prints the single vector elements to `cout`. Be sure to have read section 3.3 on how to compile STL programs.

Exercise 4.1.2: Write a STL program that takes an arbitrary sequence of binary digits (integer values 0 and 1) from `cin` and stores them into a container. When receiving a value different from `0` or `1` from `cin` stop reading. Now, you should have a container storing a sequence of `0`'s and `1`'s. After finishing the read-process, apply a "bit-stuffing" algorithm to the container. Bit-stuffing is used to transmit data from a sender to a receiver. To avoid bit sequences in the data, which would erroneously be interpreted as the stop flag (here: 01111110), it is necessary to ensure that six consecutive `1`'s in the data are splitted by inserting a `0` after each consecutive five `1`'s. **Hint:** Complexity considerations (inserting in the middle of a vector takes linear time!) and the fact, that inserting into a vector can make all iterators to elements invalid should make you choose the STL container `list`. A list of integers is defined like a `vector` by `list<int> l;` All operations explained in the vector section are provided for the list, too. Get an iterator to the first `list` element. As long as this iterator is different from the `end()` iterator increment the iterator and dereference it to get the appropriate binary value. Note that an element is always inserted before a specified iterator-position and that this insertion doesn't affect all the other iterators defined when using a `list`.

Exercise 4.1.3: Refine *Exercise 4.1.2* and print the original bit sequence and the "bit-stuffed" bit sequence to `cout`. Use the hint from *Exercise 4.1.2* to form a loop for the output procedure.

Exercise 4.1.4: Refine *Exercise 4.1.3* and print out the absolute and relative expansion of the bit sequence. The absolute expansion is the expasion measured in bits (e.g. the bit-stuffed sequence has increased by 5 bits), the relative expansion is the percentage of the expansion (e.g. the relative expansion between the "new" and "old" sequence is 5.12%).

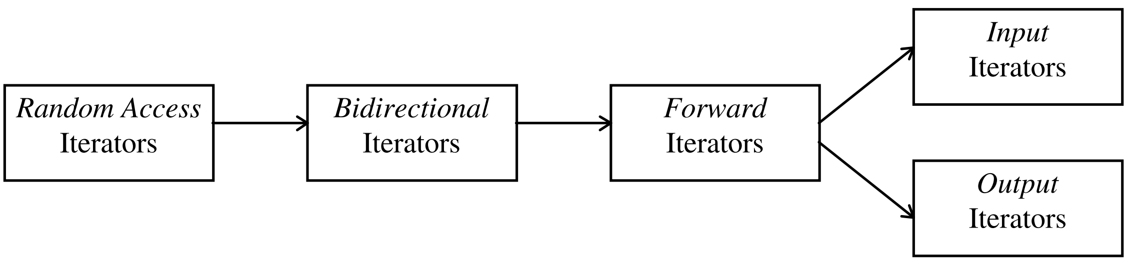
Exercise 4.1.5: Refine *Exercise 4.1.4* and write the inverse algorithm to the one in *Exercise 4.1.2* that the receiver has to perform to get the initial binary data representation. After the bit-stuffing and bit-unstuffing compare your list with the original one using the equality `operator==`. If the lists are equal, you did a fine job. **Note:** It is advisable to include a plausibility test in your unstuff algorithm. After a sequence of five consecutive ones there must be a zero, otherwise something went wrong in the stuffing algorithm.

4.2 Iterators

"Iterators are a generalization of pointers that allow a programmer to work with different data structures (containers) in a uniform manner", [2]. From the short survey in section 4.1.1 we know that iterators are objects that have `operator*` returning a value of a type called the *value type* of the iterator.

Since iterators are a generalization of pointers it is assumed that every template function that takes iterators as arguments also works with regular pointers.

There are five categories of iterators. Iterators differ in the operations defined on them. Each iterator is designed to *satisfy* a well-defined set of requirements. These requirements define what operations can be applied to the iterator. According to these requirements the iterators can be assigned to the five categories. Iterator categories can be arranged from left to right to express that the iterator category on the left satisfies the requirements of all the iterator categories on the right (and so could be called more powerful).



⟶ means, iterator category on the left satisfies the requirements of all iterator categories on the right

Figure 7: Iterator categories

This arrangement means that a template function wich expects for example a bidirectional iterator can be provided with a random access iterator, but never with a forward iterator. Imagine an algorithm that needs random access to fulfil his task, but is provided with a method that only allows to pass through the elements successively from one to the next. It simply won't work.

Iterators that point past the last element of a range are called *past-the-end* iterators. Iterators for which the `operator*` is defined are called *dereferenceable*. It is never assumed that past-the-end iterators are dereferenceable. An iterator value (i.e. an iterator of a specific iterator type) that isn't associated with a container is called *singular* (iterator) value. Pointers can also be singular. After the declaration of an uninitialized pointer with

```
int* x;
```

`x` is assumed to be singular. Dereferenceable and past-the-end iterators are always *non-singular*.

All the categories of iterators have only those functions defined that are realizeable for that category in (amortized) constant time. This underlines the efficiency concern of the library.

Because random access in a linked list doesn't take constant time (but linear time), random access iterators cannot be used with lists. Only input/output iterators up to bidirectional iterators are valid for the use with the container `list`. The following table shows the iterators that can be used with the containers `vector`, `list` and `deque` (of course all iterators that satisfy the requirements of the listed iterators can be used as well):

Container	***Iterator Category***
vector	random access iterators
list	bidirectional iterators
deque	random access iterators

Table 5: Most powerful iterator categories that can be used with vector, list and deque

Iterators of these categories are returned when using the member functions `begin` or `end` or declaring an iterator with e.g. `vector<int>::iterator i;`
The iterator categories will be explained starting with the input iterators and output iterators.

4.2.1 Input Iterators and Output Iterators

An input iterator has the fewest requirements. It has to be possible to declare an input iterator. It also has to provide a constructor. The assignment operator has to be defined, too. Two input iterators have to be comparable for equality and inequality. `operator*` has to be defined and it must be possible to increment an input iterator.

Input Iterator Requirements:

- constructor
- assignment operator
- equality/inequality operator
- dereference operator
- pre/post increment operator

Output iterators have to satisfy the following requirements:

<u>Output Iterator Requirements:</u>

- constructor
- assignment operator
- dereference operator
- pre/post increment operator

These abstract requirements should get clear if you look at special input and output iterators provided by the library - the *istream iterator* and the *ostream iterator*.

"To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like template classes are provided", [2]. These template classes are named `istream_iterator` and `ostream_iterator`. Assume we have a file filled with 0's and 1's. We want to read the values from a file and write them to `cout`. In C++ one would write:

```
ifstream[4] ifile ("example_file");
int tmp;

while (ifile >> tmp) cout[5] << tmp;
```

<u>Output (example):</u> *110101110111011*

Note: The 0's and 1's in the file have to be separated by *whitespaces* (blank, tab, newline, formfeed or carriage return).

Using an istream and an ostream iterator in combination with the algorithm `copy` enables us to write the following:

```
ifstream ifile ("example_file");

copy (istream_iterator[6]<int, ptrdiff_t> (ifile),
      istream_iterator<int, ptrdiff_t> (),
      ostream_iterator<int> (cout) );
```

The output will be the same as in the above C++ example. `copy` is an algorithm that takes two iterators to specify the range from which elements are copied and a third iterator to specify the destination where the elements should be copied to. The template function looks as follows:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last,
                     OutputIterator result);
```

The template arguments have semantic meaning, they describe the iterator categories of that iterators provided to the function at least have to be. The iterators specifying the input range have to be at least input iterators, that means that it must be possible to increment and dereference them to get the appropriate values. The iterator specifying the result position has to be at least of the output iterator category. Since forward, bidirectional and random access iterators satisfy the requirements of input *and* output iterators, they can be used instead with the same functionality.

---

[4] To use `ifstream` include `fstream.h`

[5] To use streams like `cin` and `cout` and `operator<<`, `operator>>` for streams include `iostream.h`

[6] To use `istream_iteator` or `ostream_iterator` include `iterator.h`

If you have to include `algo.h` (as in this example), `iterator.h` is already included by `algo.h`

Dereferencing an output iterator has to result in a *lvalue*, that means it has to be possible to assign a value to the dereferenced output iterator (that is as you know an object of the value type of the iterator). For output iterators, the only valid use of the `operator*` is on the left side of the assignment statement:

```
a is an output iterator, t is a value of value type T

*a = t;     valid
t = *a;     invalid
```

For output iterators, the three following conditions should hold:

- Assignment through the same value of the iterator should happen only once.

```
ostream_iterator<int> r (cout);
*r = 0;
*r = 1;
```

is not a valid code sequence.

- Any iterator value sould be assigned before it is incremented.

```
ostream_iterator<int> r (cout);
r++;
r++;
```

is not a valid code sequence.

- Any value of an output iterator may have at most one active copy at any given time.

```
// i and j are output iterators
// a and b are values written to a iterator position
i = j;
*i++ = a;
*j = b;
```

is not a valid code sequence.

For both input and output iterators algorithms working on them are assumed to be *single pass* algorithms. Such algorithms are never assumed to attempt to pass the same iterator twice.

For input iterators `r` and `s`, `r==s` does not imply `++r == ++s`:

```
ifstream ifile ("example_file")       // example_file: 0 1 2 3

istream_iterator<int, ptrdiff_t> r (ifile);
istream_iterator<int, ptrdiff_t> s (ifile);

(r==s) ? cout << "equal" : cout << "not equal";
cout << endl;

++r;
++s;

cout << *r << endl;
cout << *s << endl;

(r==s) ? cout << "equal" : cout << "not equal";
cout << endl;
```

Output: *equal*
2
3
equal

Note: For input two input iterators `a` and `b`, `a == b` implies `*a == *b`. For istream iterators, this condition doesn't hold.

When incrementing an input iterator, a value is read from the input stream and stored temporarily in the input iterator object. Dereferencing the input iterator returns the value stored.

The constructor of the istream iterator takes an input stream as its argument from which values are read. To yield an end-of-stream iterator which represents the end of file (`EOF`) of the input stream, the default constructor has to be used. To successfully construct an istream iterator, two template arguments have to be provided, too. The first argument specifies the type of the elements read from the input stream, the second is `ptrdiff_t`, that is the type of the difference of two pointers in the actual memory model (see section 4.5 - allocators).

The constructor of the ostream iterator can take one or two arguments. However, the first argument specifies the output stream to which values are written. The alternative second argument is a string which is printed between the written values. `ostream_iterator` takes a template argument which determines the type of the values written to the output stream.

It will often be asked to copy elements from an input stream (e.g. a file) directly into a container:

```
vector<int> v;
ifstream ifile ("example_file");

copy (istream_iterator<int, ptrdiff_t> (ifile),
      istream_iterator<int, ptrdiff_t> (),
      back_inserter(v) );
```

The function `back_inserter` returns a `back_insert_iterator`. This is a so-called iterator *adaptor* (explained in detail in section 4.4) and is a kind of past-the-end iterator to the container. The container, for which a back insert iterator is to be created, has to be handed over to `back_inserter`. When a value is written to the back insert iterator, it is appended to the specified container as its last element. If, for example, `v.end()` is used instead of the back insert iterator in the example above, all the values inserted will be written to the same vector position (`v.end()`), because `v.end()` isn't incremented after writing to it. This increment is internally done by the back insert iterator by calling the container member function `push_back`.

4.2.2 Forward Iterators

Forward iterators have to satisfy the following requirements:

Forward Iterator Requirements:

- constructor
- assignment operator
- equality/inequality operator
- dereference operator
- pre/post increment operator

The difference to the input and output iterators is that for two forward iterators `r` and `s`, `r==s` implies `++r==++s`. A difference to the output iterators is that `operator*` is also valid on the left side of the assignment operator (`t = *a` is valid) and that the number of assignments to a forward iterator is not restricted.

So, *multi-pass one-directional* algorithms can be implemented on containers that allow the use of forward iterators (look at *Table 5*). As an example for a single-pass one-directional algorithm `find_linear` is presented. It iterates through the elements of a container and returns the iterator position where a value provided to `find_linear` is found, otherwise the past-the-end iterator is returned. The overhead of `find_linear` is statistically *n/2*.

```
template<class ForwardIterator, class T>
ForwardIterator find_linear (ForwardIterator first,
                             ForwardIterator last, T& value) {
   while (first != last) if (*first++ == value) return first;
   return last;
}
```

```
vector<int> v (3, 1);
v.push_back (7);  // vector v: 1 1 1 7

vector<int>::iterator i = find_linear (v.begin(), v.end(), 7);
if (i != v.end() ) cout << *i; else cout << "not found";
```

Output: *7*

4.2.3 Bidirectional Iterators

In addition to forward iterators, bidirectional iterators satisfy the following requirements:

Bidirectional Iterator Requirements (additional to forward iterators'):

- pre/post decrement operator

Bidirectional iterators allow algorithms to pass through the elements forward and backward.

```
list<int> l (1, 1);
l.push_back (2);  // list l: 1 2

list<int>::iterator first = l.begin();
list<int>::iterator last  = l.end();

while (last != first) {
  --last;
  cout << *last << " ";
}
```

Output: *2 1*

The bubble sort algorithm serves as an example for a multi-pass algorithm using bidirectional iterators.

```
template <class BidirectionalIterator, class Compare>
void bubble_sort (BidirectionalIterator first, BidirectionalIterator last,
                  Compare comp)
{
  BidirectionalIterator left_el = first, right_el = first;
  right_el++;

  while (first != last)
  {

    while (right_el != last) {
      if (comp(*right_el, *left_el)) iter_swap (left_el, right_el);
      right_el++;
      left_el++;
    }
    last--;
    left_el = first, right_el = first;
    right_el++;
  }
}
```

The binary function object `Compare` has to be provided by the user of `bubble_sort`. `Compare`, which implements a binary predicate, takes two arguments and returns the result (`true` or `false`) of the predicate provided with the two arguments.

```
list<int> l;
// fill list
bubble_sort (l.begin(), l.end(), less<int>() );    // sort ascendingly
bubble_sort (l.begin(), l.end(), greater<int>() );// sort descendingly
```

4.2.4 Random Access Iterators

In addition to bidirectional iterators, random access iterators satisfy the following requirements:

Random Access Iterator Requirements (additional to bidirectional iterators'):

- operator+ (int)
- operator+= (int)
- operator- (int)
- operator-= (int)
- operator- (random access iterator)
- operator[] (int)
- operator < (random access iterator)
- operator > (random access iterator)
- operator >= (random access iterator)
- operator <= (random access iterator)

Random access iterators allow algorithms to have random access to elements stored in a container which has to provide random access iterators, like the vector.

```
vector<int> v (1, 1);
v.push_back (2); v.push_back (3); v.push_back (4); // vector v: 1 2 3 4

vector<int>::iterator i = v.begin();
vector<int>::iterator j = i + 2; cout << *j << " ";
i += 3; cout << *i << " ";
j = i - 1; cout << *j << " ";
j -= 2;
cout << *j << " ";
cout << v[1] << endl;
(j < i) ? cout << "j < i" : cout << "not (j < i)"; cout << endl;
(j > i) ? cout << "j > i" : cout << "not (j > i)"; cout << endl;
i = j;
i <= j && j <= i ? cout << "i and j equal" : cout << "i and j not equal";
cout << endl;
j = v.begin();
i = v.end();
cout << "iterator distance end - begin =^ size: " << (i - j);
```

Output: *3 4 3 1 2*
j < i
not (i > j)
i and j equal
iterator distance end - begin =^ size: 4

An algorithm that needs random access to container elements to work with O(*ld n*) is the binary search algorithm. In section 4.3 algorithms and function objects are explained and it is shown how they work together in a very advantageous way.

4.2.5 Exercises

Exercise 4.2.1: Refine *Exercise 4.1.5* by reading the original bit sequence out of a user built file *bit_seq*. Additionally, store the bit-stuffed bit sequence in the file *bit_stff* (note that the integer values in the input and output stream have to be separated by whitespaces). **Hint:** The output file *bit_stff* has to be declared as `ofstream`, which is defined like `ifstream` in `fstream.h`.

4.3 Algorithms and Function Objects

All the algorithms provided by the library are parametrized by iterator types and are so seperated from particular implementations of data structures. Because of that they are called *generic* algorithms.

4.3.1 How to create a generic algorithm

I want to evolve a generic *binary search* algorithm out of a conventional one. The starting point is a C++ binary search algorithm which takes an integer array, the number of elements in the array and the value searched for as arguments. `binary_search` returns a constant pointer to the element - if found - the *nil* pointer else.

```
const int* binary_search (const int* array, int n, int x) {

   const int* lo = array, *hi = array + n, *mid;
   while(lo != hi) {
      mid = lo + (hi - lo) / 2;
      if (x == *mid) return mid;
      if (x < *mid) hi = mid; else lo = mid + 1;
   }
   return 0;
}
```

Let us look at the assumptions this algorithm makes about its environment. `binary_search` only works with integer arrays. To make it work with arrays of arbitrary types we transform `binary_search` in a template function.

```
template<class T>
const T* binary_search (const T* array, int n, const T& x) {

   const T* lo = array, *hi = array + n, *mid;
   while(lo != hi) {
      mid = lo + (hi - lo) / 2;
      if (x == *mid) return mid;
      if (x < *mid) hi = mid; else lo = mid + 1;
   }
   return 0;
}
```

Now the algorithm is designed for use with arrays of different types. In case of not finding the value searched for, a special pointer - *nil* - is returned. This requires that such a value exists. Since we don't want to make this assumption, in case of an unsuccessful search we return the pointer `array + n` (yes, a past-the-end pointer) instead.

```
template<class T>
const T* binary_search (const T* array, int n, const T& x) {

   const T* lo = array, *hi = array + n, *mid;
   while(lo != hi) {
      mid = lo + (hi - lo) / 2;
      if (x == *mid) return mid;
      if (x < *mid) hi = mid; else lo = mid + 1;
   }
   return array + n;
}
```

Instead of handing over `array` as pointer to the first element and a size, we could also specify a pointer to the first and past the last element to approach STL's iterator concept.

```
template<class T>
const T* binary_search (T* first, T* last, const T& value) {

   const T* lo = array, *hi = array + n, *mid;
   while(lo != hi) {
      mid = lo + (hi - lo) / 2;
      if (value == *mid) return mid;
      if (value < *mid) hi = mid; else lo = mid + 1;
   }
   return last;
}
```

To specify a pointer to the end of a container instead of handing over its size has the advantage that it has not to be possible to compute `last` out of `first` with `first+n`. This is important for containers that don't allow random access to their elements. Because our `binary_search` needs random access to the elements of the container, this is of little importance in our example. Another advantage is that the *difference* type (here `int`) doesn't have to be explicitly handed over, so the user of `binary_search` doesn't even have to know it. The difference type is the type which is used to express the type of the difference of two arbitrary iterators (pointers), for example `last - first` could be of the type `signed long`.

The last step to fully adapt the algorithm to the STL style is to change the first and last pointer type from pointers to the value type to an appropriate iterator type. By this step, the information of how the algorithm steps from one element to the next is torn away from the algorithm implementation and is hidden in the iterator objects. Now, no assumptions about the mechanism to iterate through the elements are made. This mechanism is handed over to the algorithm by the iterator objects. So, the algorithm is separated from the container it works on, all the operations that deal with iterators are provided by the iterator objects themselves.

Since `binary_search` needs random access to the elements of the container it is called for and so iterators handed over to `binary_search` have to satisfy the requirements of random access iterators, we name the type of `first` and `last` "`RandomAccessIterator`":

```
template<class RandomAccessIterator, class T>
RandomAccessIterator binary_search (RandomAccessIterator first,
                                    RandomAccessIterator last,
                                    const T& value) {

   RandomAccessIterator not_found = last, mid;
   while(first != last) {
      mid = first + (last - first) / 2;
      if (value == *mid) return mid;
      if (value < *mid) last = mid; else first = mid + 1;
   }
   return not_found;
}
```

The only assumptions the algorithm makes are the random access to elements of type `T` between the two iterators (pointers) `first` and `last` and that `operator==` and `operator<` are defined for type `T` and the value type of the iterator.

This generic binary search algorithm hasn't lost anything of its functionality, especially not when dealing with built in types.

```
int x[10];                // array of ten integer values
int search_value;         // value searched for

// initialize variables

int* i = binary_search (&x[0], &x[10], search_value);
if (i == &x[10]) cout << "value not found"; else cout << "value found";
```

All the STL algorithms are constructed like our example algorithm - they try to make as few assumptions as possible about the environment they are run in.

4.3.2 The STL algorithms

The algorithms delivered with the library are divided into four groups:

group	*algorithm type*
1	mutating sequence operations
2	non-mutating sequence operations
3	sorting and related operations
4	generalized numeric operations

Table 6: STL algorithm types

Group 1 contains algorithms which don't change (mutate) the order of the elements in a container, this has not to be true for algorithms of group 2.

The algorithm `for_each` of group 1 takes two iterators and a function `f` of type `Function` as arguments:

```
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function f);
```

The template argument `f` of type `Function` must not be mixed up with a "pure" C++ function, because such a function can only be used in a roundabout way (see section 4.4.3). The template function `for_each` expects a *function object* (section 2.2) as argument. `f` is assumed not to apply any non-constant function through the dereferenced iterator.

`for_each` applies f to the result of dereferencing every iterator in the range `[first, last)` and returns `f`. If `f` returns a value, it is ignored. The following example computes the sum of all elements in the range `[first, last)`.

```
template <class T>
class sum_up {
   public:
      void operator() (const T& value) { sum += value; }
      const T& read_sum() { return sum; }
  private:
      static T sum;
};

int sum_up<int>::sum;

void main(void) {

   deque[7]<int> d (3,2);
   sum_up<int> s;
   for_each (d.begin(), d.end(), s);
   cout << s.read_sum();
}
```

<u>Output:</u> *6*

Group 1 also contains an algorithm `find`, which is very similar to `find_linear` from section 4.2.2.

[7] To use a `deque` include `deque.h`

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last,
                   const T& value);
```

`find` takes a range and a reference to a value of arbitrary type. It assumes that `operator==` for the value type of the iterator and `T` is defined. Additionally to `find` an algorithm named `find_if` is provided, which takes a predicate `pred` of type `Predicate`.

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

`find_if` (like `find`) returns the first iterator `i` in the range `[first, last)`, for which the following condition holds: `pred(*i) = true`. If such an iterator doesn't exist, a past-the end iterator is returned.

```
template <class T>
class find_first_greater {
  public:
     find_first_greater() : x(0) {}
     find_first_greater(const& xx) : x(xx) {}
     int operator() (const T& v) { return v > x; }
  private:
     T x;
};

vector<int> v;
// fill vector with 1 2 3 4 5
vector<int>::iterator i = find_if (v.begin(), v.end(),
                                   find_first_greater<int> (3));
i != v.end()? cout << *i : cout << "not found";
```

<u>Output:</u> *4*

Generally, if there is a version of an algorithm which takes a predicate, it gets the name of the algorithm with the suffix `_if`.

Some algorithms, like `adjacent_find`, take a binary predicate `binary_pred` of type `BinaryPredicate`. `adjacent_find` returns the first iterator `i`, for which the following condition holds: `binary_pred (*i, *(i+1)) == true`.

```
template <class InputIterator, class BinaryPredicate>
InputIterator adjacent_find(InputIterator first, InputIterator last,
                            BinaryPredicate binary_pred);
```

For example, if you want to find the first pair of values, whose product is odd, you could write this:

```
template <class T>
class prod_odd {
  public:
     int operator() (const T& v1, const T& v2)
            { return v1%2 != 0 && v2%2 != 0; }
};

list<int> l;
// fill list with 2 9 6 13 7
list<int>::iterator i = adjacent_find (l.begin(), l.end(),
                                       prod_odd<int>());
if (i != l.end()) { cout << *i << " "; i++; cout << *i++; }
else cout << "not found";
```

Output: *13 7*

Algorithms can work *in place*, that means they do their work within the specified range. Some algorithms have an additional version which copies well-defined elements to an output iterator `result`. When such a version is provided, the algorithm gets the suffix `_copy` (which precedes a probable suffix `_if`). For example there is `replace_copy_if`, which assigns to every iterator in the range `[result, result+(last-first) )` either a new value (which has to be specified) or the original value. This depends on a predicate given as argument.

```
template <class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
                               OutputIterator result, Predicate pred,
                               const T& new_value);
```

All the operations in group 3 have two versions. One that takes a function object `comp` of type `Compare` and another that uses `operator<` to do the comparison. `operator<` and `comp`, respectively, have to induce a total ordering relation on the values to ensure that the algorithms work correctly.

```
vector<int> v;
// fill v with 3 7 5 4 2 6
sort (v.begin(), v.end() );
sort (v.begin(), v.end(), less<int>() );
sort (v.begin(), v.end(), greater<int>() );
```

Output: *2 3 4 5 6 7*
2 3 4 5 6 7
7 6 5 4 3 2

Since the library provides function objects for all of the comparison operators in the language we can use `less` to sort the container ascendingly and `greater` to sort it descendingly.
All the provided function objects are derived either from `unary_function` or from `binary_function` to simplify the type definitions of the argument and result types.

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};
```

```
template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

STL provides function objects for all of the arithmetic operations in the language. `plus`, `minus`, `times`, `divides` and `modulus` are binary operations whereas `negate` is a unary operation. As examples, look at `plus` and `negate`, the other functions objects are defined accordingly.

```
template <class T>
struct plus : binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x + y; }
};
```

```
template <class T>
```

```
struct negate : unary_function<T, T> {
    T operator()(const T& x) const { return -x; }
};
```

The mentioned comparison function objects are `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` and `less_equal`, they are all binary function objects. `less` shall serve as example.

```
template <class T>
struct less : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x < y; }
};
```

Additionally, the binary function objects `logical_and` and `logical_or` exist, `logical_not` is a unary function object.

```
template <class T>
struct logical_and : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x && y; }
};
```

```
template <class T>
struct logical_not : unary_function<T, bool> {
    bool operator()(const T& x) const { return !x; }
};
```

The rest of the function object implementations can be found in [2], 6.

In group 4, the algorithm `accumulate` takes a binary operation `binary_op` of type `BinaryOperation`. The algorithm `accumulate` does the same as `for_each` used with the function object `sum_up` (presented earlier in this section).

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);
```

For each iterator `i` in `[first, last)`, `acc = acc + *i` is computed, then `acc` is returned. `acc` can be initialized with a starting value. Instead of `operator+`, an arbitrary binary operation can be defined by the user, or a STL function object can be used.

```
vector<int> v;
v.push_back (2); v.push_back (5);
cout << accumulate (v.begin(), v.end(), 10, divides<int>() );
```

<u>Output:</u> *1*

I want to present an example which implements a spell-checker. For this purpose we assume the following:

- The dictionary is stored in a file
- The text to check is stored in a file
- The words of the text should be checked against dictionary
- Every word not found or misspelled should be displayed

We decide to use a non-associative container (see section 4.1, introduction), which holds the dictionary. The dictionary is assumed to be sorted. Now, we express the spell-checker functionality in pseudo code.

```
for every word in text
   check against dictionary
   if not in dictionary write to output
```

This pseudo code can be expressed in different way:

```
copy every word of text to output
   that is not in the dictionary
```

The last pseudo code variation can more directly be translated into a STL program. Since we need a mechanism that tells us if a word is or is not in the dictionary, we encapsulate this functionality in a function object.

```
template <class bidirectional_iterator, class T>
class nonAssocFinder {
public:
   nonAssocFinder(bidirectional_iterator begin,
                  bidirectional_iterator end) :
      _begin(begin), _end(end) {}

   bool operator() (const T& word) {
      return binary_search(_begin, _end, word); }

private:
   bidirectional_iterator _begin;
   bidirectional_iterator _end;
};
```

The function object `nonAssocFinder` is initialized with the iterators `begin` and `end` that have to be at least of the bidirectional iterator category. The function call operator takes a word and returns a boolean value, which states if the word has been found in the dictionary (the type `bool` is defined by STL). This boolean value is returned by the STL algorithm `binary_search`.

```
template <class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value);
```

The first thing we do in our program is to define a dictionary as a vector of type `string` and fill it out of an input stream.

```
typedef vector<string[8]> dict_type;

ifstream dictFile("dict.txt");
ifstream wordsFile("words.txt");

dict_type dictionary;

copy (istream_iterator<string, ptrdiff_t>(dictFile),
      istream_iterator<string, ptrdiff_t>(),
      back_inserter(dictionary));
```

[8] To use the `string` type include `cstring.h`

Then we use the STL algorithm `remove_copy_if` to achieve the functionality wanted.

```
template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                  OutputIterator result, Predicate pred);
```

`remove_copy_if` writes all elements referred to by the iterator `i` in the range `[first, last)` to the output iterator `result`, for which the following condition does *not* hold: `pred(*i) == true`. The algorithm returns the end of the resulting range. The rest of the spell-checker program proves to be a single statement.

```
remove_copy_if (
  istream_iterator<string, ptrdiff_t>(wordsFile),
  istream_iterator<string, ptrdiff_t>(),
  ostream_iterator<string>(cout, "\n"),
  nonAssocFinder<dict_type::iterator,
                 dict_type::value_type>
     (dictionary.begin(), dictionary.end()));
```

`remove_copy_if` reads words from the input stream `wordsFile` and writes the words for which `nonAssocFinder` returns `false` (i.e. which are either not found or misspelled) to `cout`.

The components used are:

- algorithms: `copy` and `remove_copy_if`
- container: `vector`
- user defined: function object `nonAssocFinder`

Now you should have the basics to understand the chapter on algorithms in [2], 10. Since this document is very theoretical, the algorithms in combination with a description and examples can be found in [4], 6. A complete STL example can be found in [4], 5.

4.3.3 Exercises

Exercise 4.3.1: Fill two containers with the same number of integer values. Create a new container, whose elements are the sum of the appropriate elements in the original container. **Hint:** The library provides an algorithm and a function object to do the exercise.

Exercise 4.3.2: Write a generator object which can be used with the STL algorithm `generate` (group 2) to fill containers with certain values. It should be possible to specify a starting value and a step size, so that the first element in the container is the starting value and every further element is the sum of the preceding element and the step size.

4.4 Adaptors

As stated in [2], 11, "Adaptors are template classes that provide interface mappings". Adaptors are classes that are based on other classes to implement a new functionality. Member functions can be added or hidden or can be combined to achieve new functionality.

4.4.1 Container Adaptors

Stack. A stack can be instantiated either with a `vector`, a `list` or a `deque`. The member functions `empty`, `size`, `top`, `push` and `pop` are accessible to the user.

```
stack<vector<int> >     s1;
stack<list<int> >       s2;
stack<deque<int> >      s3;

s1.push(1); s1.push(5);
cout << s1.top() << endl;
s1.pop();
cout << s1.size() << endl;
s1.empty()? cout << "empty" : cout << "not empty";
```

(The first `stack` carries footnote marker [9].)

Output: *5*
1
not empty

`top` returns the element on the top of the stack, `pop` removes the top element from the stack. For comparison of two stacks, `operator==` and `operator<` are defined.

Queue. A queue can be instantiated with a `list` or a `deque`.

```
queue<list<int> >  q1;
queue<deque<int> > q2;
```

Its public member functions are `empty`, `size`, `front`, `back`, `push` and `pop`. `front` returns the next element from the queue, `pop` removes this element. `back` returns the last element pushed to the queue with `push`. As with the stack, two queues can be compared using `operator==` and `operator<`.

Priority queue. A priority queue can be instantiated with a `vector` or a `deque`. A priority queue holds the elements added by `push` sorted by using a function object `comp` of type `Compare`.

```
// use less as compare object
priority_queue<vector<int>, less<int> > pq1;
// use greater as compare object
priority_queue<deque<int>, greater<int> > pq2;

vector v(3, 1);

// create a priority_queue out of a vector, use less as compare object
priority_queue<deque<int>, less<int> > pq3 (v.begin(), v.end() );
```

`top` returns the element with the highest priority, `pop` removes this element. The element with the highest priority is determined by the sorting order imposed by `comp`. Note, that a priority queue internally is implemented using a heap. So, when `less` is used as compare object, the element with the highest priority `h` will be one of the elements for which the following condition holds: `less (h, x) == false` for all elements `x` in the priority queue.

Additionally, the member functions `empty` and `size` are provided. Note that no comparison operators for priority queues are provided. For the implementations of the container adaptors, read [2], 11.1.

[9] To use a `stack` , `queue` or `priority_queue` include `stack.h`

4.4.2 Iterator Adaptors

Reverse Iterators. For the bidirectional and random access iterators corresponding reverse iterator adaptors that iterate through a data structure in the opposite direction are provided.

```
list<int> l;
// fill l with 1 2 3 4
reverse_bidirectional_iterator[10]<list<int>::iterator,
                                 list<int>::value_type,
                                 list<int>::reference_type,
                                 list<int>::difference_type> r (l.end());
cout << *r << " ";
r++;
cout << *r << " ";
r --;
cout << *r;
```

Output: *4 3 4*

```
list<int> l;
// fill l with 1 2 3 4

copy (reverse_iterator<int*, int, int&, ptrdiff_t> (l.end()),
      reverse_iterator<int*, int, int&, ptrdiff_t> (l.begin()),
      ostream_iterator<int> (cout, " ") );
```

Output: *4 3 2 1*

For all the sequence containers (`vector`, `list` and `deque`) the member functions `rbegin` and `rend` are provided, which return the appropriate reverse iterators.

```
list<int> l;
// fill l with 1 2 3 4

copy (l.rbegin(), l.rend(), ostream_iterator<int> (cout, " "));
```

Output: *4 3 2 1*

Insert Iterators. A kind of iterator adaptors, called *insert iterators*, simplify the insertion into containers. The principle is that writing a value to an insert iterator inserts this value into the container out of which the insert iterator was constructed. To define the position, where the value is inserted, three different insert iterator adaptors are provided:

- `back_insert_iterator`
- `front_insert_iterator`
- `insert_iterator`

`back_insert_iterator` and `front_insert_iterator` are constructed out of a container and insert elements at the end and at the beginning of this container, respectively. A `back_insert iterator` requires the container out of which it is constructed to have `push_back` defined, a `front_insert_iterator` correspondingly requires `push_front`.

```
deque<int> d;

back_insert_iterator<deque<int> >     bi (d);
front_insert_iterator<deque<int> >    fi (d);
```

[10] To use `reverse_bidirectional_iterator` or `reverse_iterator` include `iterator.h`

insert_iterator is constructed out of a container and an iterator i, before which the values written to the insert iterator are inserted.

```
deque<int> d;
insert_iterator<deque<int> >   i (d, d.end() );
```

Insert iterators satisfy the requirements of output iterators, that means that an insert iterator can always be used when an output iterator is required. operator* returns the insert iterator itself.

The three functions back_inserter, front_inserter and inserter return the appropriate insert iterators.

```
template <class Container>
back_insert_iterator<Container> back_inserter(Container& x) {
    return back_insert_iterator<Container>(x);
}

template <class Container>
front_insert_iterator<Container> front_inserter(Container& x) {
    return front_insert_iterator<Container>(x);
}

template <class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i) {
    return insert_iterator<Container>(x, Container::iterator(i));
}
```

```
ifstream f ("example");  // file example: 1 3
deque<int> d;
copy (istream_iterator<int, ptrdiff_t>(f),
      istream_iterator<int, ptrdiff_t>(),
      back_inserter(d) );

vector<int> w (2,7);
copy (w.begin(), w.end(), front_inserter (d) );

insert_iterator<deque<int> > i = inserter (d, ++d.begin() );
*i = 9;
```

Ouptut: *7 9 7 1 3*

Raw Storage Iterator. A raw_storage_iterator enables algorithms to store results into uninitialized memory.

```
vector<int> a (2, 5);
vector<int> b (2, 7);
int *c = allocate((ptrdiff_t) a.size(), (int*)0 );

transform ( a.begin(), a.end(), b.begin(),
            raw_storage_iterator<int*, int> (c), plus<int>() );

copy (&c[0], &c[2], ostream_iterator<int> (cout, " ") );
```

Output: *12 12*

The function allocate is provided by the STL allocator (see 4.5), transform is an algorithm of group 2 (see 4.3.2). To use a raw storage iterator for a given type T, a construct function must be defined, which puts results directly into uninitialized memory by calling the appropriate copy constructor. The following construct function is provided by STL:

```
template <class T1, class T2>
inline void construct(T1* p, const T2& value) {
    new (p) T1(value);
}
```

```
int a[10] = {1, 2, 3, 4, 5};
copy (&a[0], &a[5], raw_storage_iterator<int*, int> (&a[5]) );
```

4.4.3 Function Adaptors

Negators. The negators `not1` and `not2` are functions which take a unary and a binary predicate, respectively, and return their complements.

```
template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred) {
    return unary_negate<Predicate>(pred);
}

template <class Predicate>
binary_negate<Predicate> not2(const Predicate& pred) {
    return binary_negate<Predicate>(pred);
}
```

The classes `unary_negate` and `binary_negate` only work with function object classes which have argument types and result type defined. That means, that `Predicate::argument_type` and `Predicate::result_type` for unary function objects and `Predicate::first_argument_type`, `Predicate::second_argument_type` and `Predicate::result_type` for binary function objects must be accessible to instantiate the negator classes.

```
vector<int> v;
// fill v with 1 2 3 4
sort (v.begin(), v.end(), not2 (less_equal<int>()) );
```

Output: *4 3 2 1*

Binders. "The binders `bind1st` and `bind2nd` take a function object `f` of two arguments and a value `x` and return a function object of one argument constructed out of `f` with the first or second argument correspondingly bound to `x`.", [2],11.3.2. Imagine that there is a container and you want to replace all elements less than a certain bound with this bound.

```
vector<int> v;
// fill v with 4 6 10 3 13 2
int bound = 5;

replace_if (v.begin(), v.end(), bind2nd (less<int>(), bound), bound);

// v: 5 6 10 5 13 5
```

`bind2nd` returns a unary function object `less` that takes only one argument, because the second argument has previously been bound to the value `bound`. When the function object is applied to a dereferenced iterator `i`, the comparison `*i < bound` is done by the function-call operator of `less`.

Adaptors for pointers to functions. The STL algorithms and adaptors are designed to take function objects as arguments. If a usual C++ function shall be used, it has to be wrapped in a function object.

The function `ptr_fun` takes a unary or a binary function and returns the corresponding function object. The function-call operator of these function objects simply calls the function with the arguments provided.
For example, if a vector of character pointers is to be sorted lexicographically with respect to the character arrays pointed to, the binary C++ function `strcmp` can be transformed into a comparison object and can so be used for sorting.

```
vector<char*> v;
char* c1 = new char[20]; strcpy (c1, "Tim");
char* c2 = new char[20]; strcpy (c2, "Charles");
char* c3 = new char[20]; strcpy (c3, "Aaron");
v.push_back (c1); v.push_back (c2); v.push_back (c3);

sort (v.begin(), v.end(), ptr_fun (strcmp) );
copy (v.begin(), v.end(), ostream_iterator<char*> (cout, " ") );
```

Output: *Aaron Charles Tim*

Note: The above example causes memory leaks, because the memory allocated with `new` is not automatically deallocated. See section 4.5 for a solution.

4.5 Allocators and memory handling

"One of the common problems in portability is to be able to encapsulate the information about the memory model.", [2], 7. This information includes the knowledge of

- pointer types
- type of their difference (difference type `ptrdiff_t`)
- type of the size of objects in a memory model (size type `size_t`)
- memory allocation and deallocation primitives.

STL provides allocators which are objects that encapsulate the above information. As mentioned in section 4.1.1, all the STL containers are parametrized in terms of allocators. So, containers don't have any memory model information coded inherently but are provided with this information by taking an allocator object as argument.

The idea is that changing memory models is as simple as changing allocator objects. The allocator `allocator`, which is defined in `defalloc.h`, is used as default allocator object. The compiler vendors are expected to provide allocators for the memory models supported by their product. So, for Borland C++ allocators for different memory models are provided (see 3.2).

For every memory model there are corresponding `allocate`, `deallocate`, `construct` and `destroy` template functions defined. `allocate` returns a pointer of type `T*` to an allocated buffer, which is no less than `n*sizeof(T)`.

```
template <class T>
inline T* allocate(ptrdiff_t size, T*);
```

`deallocate` frees the buffer allocated by `allocate`.

```
template <class T>
inline void deallocate(T* buffer);
```

`construct` puts results directly into uninitialized memory by calling the appropriate copy constructor.

```
template <class T1, class T2>
inline void construct(T1* p, const T2& value) {
    new (p) T1(value);
}
```

`destroy` calls the destructor for a specified pointer.

```
template <class T>
inline void destroy(T* pointer) {
    pointer->~T();
}
```

If you have a container of pointers to certain objects, the container destructor calls a special destroy function to call all the single pointer destructors and free the memory allocated. To make this work under Borland C++, a template specialization must be provided.

```
class my_int {
public:
   my_int (int i = 0)  { ii = new int(i); }
  ~my_int () { delete ii; }
private:
   int* ii;
};

// the following template specialization is necessary when using Borland C++

inline void destroy (my_int** pointer) {
  (*pointer)->~my_int();
}

void main (void) {

   vector<my_int*> v (10);
   for (int i = 0; i < 10; i++) { v[i] = new my_int(i); }

   // allocated my_int memory and vector v are destroyed at end of scope
}
```

When you use a container of pointers to objects which do not have an explicit destructor defined, a function like `seq_delete` can be implemented to free all the memory allocated.

```
template <class ForwardIterator>
inline void seq_delete (ForwardIterator first, ForwardIterator last) {

   while (first != last) delete *first++;
}

vector<char*> v;
char* c1 = new char[20]; strcpy (c1, "Tim");
char* c2 = new char[20]; strcpy (c2, "Charles");
v.push_back (c1); v.push_back (c2);

seq_delete (v.begin(), v.end() );

// vector v is destroyed at the end of scope
```

5 The remaining STL components

The remaining STL components and topics not dealt with yet will be described here.

5.1 How components work together

To make it clear how all STL components work together the relations between the components are topic of this section.

Containers store objects of arbitrary types. Containers are parametrized by allocators. Allocators are objects which encapsulate information about the memory model used. They provide memory primitives to handle memory accesses uniformly. Every memory model has its characteristic, tailor-made allocator. Containers use allocators to do their memory accesses. A change of the memory model used leads to a change of the allocator object given as an argument to the container. This means, that on the code level a container object is invariant under different memory models.
An algorithm is a computation order. So, two algorithms should differ in the computations done by them, not in the access method used to read input data and write output data. This can be achieved when data is accessed in a uniform manner. STL provides a uniform data access mechanism for its algorithms - iterators. Different iterators provide different access modes. The basic input and output unit is the range, which is a well-defined sequence of elements.
Function objects are used in combination with algorithms to encapsulate, for example, predicates, functions and operations to extend the algorithms' utility.
Adaptors are interface mappings, they implement new objects with different or enhanced functionality on the basis of existing components.
It has to be said that this decomposition of the component space is arbitrary to a certain extent but designed to be as orthogonal as possible. This means that interferences between components are reduced as far as possible.

The clean, orthogonal and transparent design of the library shall help to

- simplify application design and redesign
- decrease the lines of code to be written
- increase the understandability and maintainability
- provide a basis for standard certifying and quality assurance as in other areas of system architecture, design and implementation.

5.2 Vector

Additionally to the member functions described in section 4.1.1, a `reserve` member function is provided, which informs the vector of a planned change in size. This enables the vector to manage the storage allocation accordingly. `reserve` does not change the size of the vector and reallocation happens if and only if the current capacity is less than the argument of `reserve`.

```
void reserve(size_type n);
```

After a call of `reserve`, the capacity (i.e. the allocated storage) of the vector is greater or equal to the argument of `reserve` if reallocation has happened, equal to its previous value otherwise. This means, that if you use `reserve` with a value greater than the actual value of capacity, reallocation happens and afterwards, the capacity of the vector is greater or equal to the value given as argument to `reserve`.

To make it clear, why such a member function is provided, remember that reallocation invalidates all the references, pointers and iterators referring to the elements in the sequence. The use of `reserve` guarantees that no reallocation takes place during the insertions that happen after a call of `reserve` until the time when the size of the vector reaches the capacity caused by the call of `reserve`.
With this in mind, take a look at exercise 4.1.1. We decided to use a list for storing the single "bits", because inserting into a list never invalidates any of the iterators to this container, which was essential for the bit-stuff algorithm to work. Now, knowing of the existence of `reserve`, we can use this member function to reserve a certain vector capacity and are so in a position to use a vector as well. After the call of `reserve`, we can insert elements into the vector till `capacity` is reached being sure that no reallocation will happen. The argument `n` of `reserve` has to be computed by considering a maximum number of bits to be bit-stuffed and the worst case expansion, which happens when bit-stuffing a sequence only consisting of `1`'s.

5.3 List

Unlike a vector, a list doesn't provide random access to its elements. So, the member functions `begin`, `end`, `rbegin` and `rend` return bidirectional iterators. In addition to the member functions `push_back` and `pop_back`, `list` provides `push_front` and `pop_front` to add and remove an element at its beginning, because these operations can be done in constant time.

The container `list` provides special mutative operations. It is possible to splice two lists into one (member function: `splice`), that is to insert the content of one list before an iterator position of another. Two lists can be merged (`merge`) into one using `operator<` or a compare function object, a list can be reversed (`reverse`) and sorted (`sort`). It is also possible to remove all but first element from every consecutive group of equal elements (`unique`).
For an exact description of all these member functions read [2], 8.1.2.

5.4 Deque

As a vector, a deque supports random access iterators. But in addition to the vector, which only allows constant time insert and erase operations at the end, a deque supports the constant time execution of these operations at the end as well as at the *beginning*. Insert and erase in the middle take constant time.
Because of these constant insert and erase operations at the beginning, a deque provides the member functions `push_front` and `pop_front`. Note, that `insert`, `push`, `erase` and `pop` invalidate all the iterators and references to the deque.
Further information concerning the deque can be found in [2], 8.1.3.

5.5 Iterator Tags

Every iterator `i` must have an expression `iterator_tag(i)` defined, which returns the *most specific* category tag that describes its behaviour.

The available iterator tags are: `input_iterator_tag, output_iterator_tag, forward_iterator_tag, bidirectional_iteerator_tag, random_access_iterator_tag`.

The most specific iterator tag of a built in pointer would be the random access iterator tag.

```
template <class T>
```

```
inline random_access_iterator_tag iterator_category (const T*) {
   return random_access_iterator_tag();
}
```

A user defined iterator which satisfies, for example, the requirements of a bidirectional iterator can be included into the bidirectional iterator category.

```
template <class T>
inline bidirectional_iterator_tag iterator_category (const MyIterator<T>&)
{
   return bidirectional_iterator_tag();
}
```

Iterator tags are used as "compile time tags for algorithm selection", [2], 5.6. They enable the compiler to use the most efficient algorithm at compile time.

Imagine the template function `binary_search` which could be designed to work with bidirectional iterators as well as with random access iterators. To use the tag mechanism, the two algorithms should be implemented as follows:

```
template<class BidirectionalIterator, class T>
BidirectionalIterator binary_search (BidirectionalIterator first,
                                     BidirectionalIterator last,
                                     const T& value,
                                     bidirectional_iterator_tag) {
   // more generic, but less efficient algorithm
}
```

```
template<class RandomAccessIterator, class T>
RandomAccessIterator binary_search (RandomAccessIterator first,
                                    RandomAccessIterator last,
                                    const T& value,
                                    random_access_iterator_tag) {
   // more efficient, but less generic algorithm
}
```

To use binary_search, a kind of stub function has to be written:

```
template<class BidirectionalIterator, class T>
inline BidirectionalIterator binary_search (BidirectionalIterator first,
                                            BidirectionalIterator last,
                                            const T& value) {

   binary_search (first, last, value, iterator_category(first));
}
```

At compile time, the compiler will choose the most efficient version of `binary_search`. The tag mechanism is fully transparent to the user of `binary_search`.

5.6 Associative Containers

"Associative containers provide an ability for fast retrieval of data based on keys.", [2], 8.2. Associative containers, like sequence containers, are used to store data. But in addition to that associative containers are designed with an intention to optimize the retrieval of data by organizing the single data records in a specialized structure (e.g. in a tree) using keys for identification. The library provides four different kinds of associative containers: `set`, `multiset`, `map` and `multimap`.
`set` and `map` support *unique keys*, that means that those containers may contain at most one element (data record) for each key. `multiset` and `multimap` support *equal keys*, so more than

one element can be stored for each key. The difference between `set` (`multiset`) and `map` (`multimap`) is that a `set` (`map`) stores data which inherently contains the key expression. `map` (`multimap`) stores the key expression and the appropriate data separately, i.e. the key has not to be part of the data stored.

Imagine we have objects that encapsulate the information of an employee at a company. An employee class could look like this:

```
class employee_data {
public:
   employee_data() : name (""), skill(0), salary(0) {}
   employee_data(string n, int s, long sa) :
      name (n), skill (s), salary (sa) {}

   string   name;
   int      skill;
   long     salary;

   friend ostream& operator<< (ostream& os, const employee_data& e);
};

ostream& operator<< (ostream& os, const employee_data& e) {
   os << "employee: " << e.name << " " << e.skill << " " << e.salary;
   return os;
}
```

If we want to store employee data in a `set` (`multiset`), the key has to be included in the object stored:

```
class employee {
public:
   employee (int i, employee_data e) :
      identification_code (i), description (e) {}

   int identification_code;     // key expression to identify an employee
   employee_data description;

   bool operator< (const employee& e) const {
     return identification_code < e.identification_code; }
};
```

Now we are able to declare a `set` (`multiset`) of employees:

```
set[11] <employee, less<employee> > employee_set;

multiset[12] <employee, less<employee> > employee_multiset;
```

Using a `set` (`multiset`), `employee` is both the *key type* and the *value type* of the set (`multiset`).

All associative containers are parametrized on a class `Key`, which is used to define `key_type`, and a so-called *comparison object* of class `Compare`, for example:

[11] To use a set include `set.h`
[12] To use a multiset include `multiset.h`

```
template <class Key, class Compare = less<Key>,
          template <class U> class Allocator = allocator>
class set {
   typedef Key key_type;
   typedef Key value_type;
    ...
};
```

If we want to store employee data in a `map` (`multimap`), the key type is `int` and the value type is `pair<const int, employee_data>`:

```
map[13] <int, employee_data, less<int> > employee_map;

multimap[14] <int, employee_data, less<int> > employee_multimap;
```

```
template <class Key, class T, class Compare = less<Key>,
          template <class U> class Allocator = allocator>
class map {
   typedef Key key_type;
   typedef pair<const Key, T> value_type;
    ...
};
```

Two keys `k1` and `k2` are considered to be equal if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`, so equality is imposed by the comparison object and not by `operator==`.

The member function `key_comp` returns the comparison object out of which the associative container has been constructed. `value_comp` returns an object constructed out of the comparison object to compare values of type `value_type`. All associative containers have the member functions `begin`, `end`, `rbegin`, `rend`, `empty`, `size`, `max_size` and `swap` defined. These member functions are equivalent to the appropriate sequence container member functions. An associative container can be constructed by specifying no argument (`less<Key>` is used as default comparison object) or by specifying a comparison object. It can be constructed out of a sequence of elements specified by two iterators or another associative container. `operator=` (assignment operator) is defined for all associative containers. Associative containers provide bidirectional iterators.

Now we want to store some employee data in the set. We can use the `insert` member function:

```
employee_data ed1 ("john", 1, 5000);
employee_data ed2 ("tom", 5, 2000);
employee_data ed3 ("mary", 2, 3000);

employee e1 (1010, ed1);
employee e2 (2020, ed2);
employee e3 (3030, ed3);

pair<set <employee, less<employee> >::iterator, bool>
   result = employee_set.insert (e1);

if (result.second) cout << "insert ok"; else cout << "not inserted";
cout << endl << (*result.first).description.name << endl;

result = employee_set.insert (e1);
if (result.second) cout << "insert ok"; else cout << "not inserted";
```

[13] To use a map include `map.h`
[14] To use a multimap include `multimap.h`

```
pair<map <int, employee_data, less<int> >::iterator, bool>
   result1 = employee_map.insert (make_pair (1010, ed1));

multiset <employee, less<employee> >::iterator
   result2 = employee_multiset.insert (e1);

multimap <int, employee_data, less<int> >::iterator
  result3 = employee_multimap.insert (make_pair (1010, ed1));
```

Output: *insert ok*
john
not inserted

Note: For users of Borland C++ it has to be said that the above map and multimap insert operations can only be compiled with a change in the code in `map.h` and `multimap.h`. Instead of "`typedef pair<const Key, T> value_type`" I used "`typedef pair<Key, T> value_type`".

`insert` takes an object of type `value_type` and returns a pair consisting of an iterator and a bool value. The bool value indicates whether the insertion took place. In case of an associative container supporting unique keys, the iterator points to the element with the key equal to the key of the element specified as argument, in case of an associative container supporting equal keys to the newly inserted element. `insert` does not affect the validity of iterators and references to the container.

A second version of `insert` takes a range specified by two iterators and inserts the appropriate elements into the associative container (the return value is `void`):

```
pair<int, employee_data> a[2]  = { make_pair (2020, ed2),
                                   make_pair (3030, ed3) };
employee_map.insert (&a[0], &a[2]);
```

The `find` member function takes a key value and returns an iterator, which indicates the success of the search operation:

```
map <int, employee_data, less<int> >::const_iterator i
   = employee_map.find (3030);

if (i == employee_map.end() ) cout << "not found";
else cout << (*i).second.name;
```

Output: *mary*

`map` is the only associative container with provides the subscribe operator (`oprator[]`) to address elements directly:

```
employee_data d = employee_map[2020];
cout << d;
```

Output: *tom 5 2000*

The `erase` member function can take a value of type `key_type`, a single iterator or a range specifying the element or elements to be erased:

```
employee_map.erase (3030);
employee_map.erase (employee_map.begin() );
employee_map.erase (employee_map.begin(), employee_map.end() );
```

```
if (employee_map.empty() ) cout << "employee_map is empty";
```

Output: *employee_map is empty*

`erase` invalidates only the iterators and references to the erased elements.

Since it doesn't make sense to store more than one employee under an employee key, for the demonstration of an associative container supporting equal keys a slightly different example is used. A number of employees is stored under the same key which represents a department code. We can use the `employee_multimap` container declared earlier in this section:

```
// employee_multimap is empty
employee_multimap.insert (make_pair(101, ed1));    // department code 101
employee_multimap.insert (make_pair(101, ed2));
employee_multimap.insert (make_pair(102, ed3));    // department code 102
```

`count` takes a key value and returns the number of elements stored under this key value.

```
multimap <int, employee_data, less<int> >::size_type count
   = employee_multimap.count (101);
cout << count;
```

Output: *2*

`lower_bound (k)` with `k` of type `key_type` returns an iterator pointing to the first element with key not less than `k`. `upper_bound (k)` returns an iterator pointing to the first element with key greater than `k`. `equal_range (k)` returns a pair of iterators with the first iterator being the return value of `lower_bound (k)` and the second being the return value of `upper_bound (k)`.

```
ostream& operator<< (ostream& os, const pair<int, employee_data>& p) {
   os << "employee: " << p.second.name << " " << p.second.skill << " " <<
           p.second.salary;
   return os;
}

typedef multimap <int, employee_data, less<int> >::iterator j;

pair<j, j> result = employee_multimap.equal_range (101);

copy (result.first,
      result.second,
      ostream_iterator<pair<int, employee_data> > (cout , "\n") );
```

Output: *john 1 5000*
tom 5 2000

6 Copyright

7 Literature

[1] Stroustrup, Bjarne: The C++ programming language -- 2nd ed.
June, 1993

[2] Lee, Meng; Stepanov, Alex: The Standard Template Library
HP Labaratories, 1501 Page Mill Road, Palo Alto, CA 94304
February 7, 1995

[3] STL++
The Enhanced Standard Template Library, Tutorial & Reference Manual
Modena Software Inc., 236 N. Santa Cruz Ave, Suite 213, Los Gatos CA 95030
1994

[4] Standard Template Library Reference
Rensselaer Polytechnic Institute, 1994
includes as chapter 6
The STL Online Algorithm Reference
Cook, Robert Jr.; Musser, David R.; Zalewski, Kenneth J.
online at http://www.cs.rpi.edu/~musser/stl.html