Part I
Introduction

The concept of the C++ Standard Template Library

1

Summary: *There are several libraries for containers and algorithms in C++. These libraries are not standardized and are not interchangeable. In the course of the now finished standardization of the C++ programming language, a template-based library for containers and optimized algorithms has been incorporated into the standard. This chapter explains the concept of this library and describes it with the aid of some examples.*

The big advantage of templates is plain to see. Evaluation of templates is carried out at compile time, there are no run time losses – for example, through polymorph function access in case genericity is realized with inheritance. The advantage of standardization is of even greater value. Programs using a standardized library are more portable since all compiler producers will be oriented towards the standard. Furthermore, they are easier to maintain since the corresponding know-how is much more widespread than knowledge of any special library.

The emphasis is on *algorithms* which cooperate with *containers* and *iterators* (Latin *iterare* = repeat). Through the template mechanism of C++, containers are suited for objects of the most varied classes. An iterator is an object which can be moved on a container like a pointer, to refer either to one or another object. Algorithms work with containers by accessing the corresponding iterators. The concepts will be presented in more detail later.

References: Owing to its very nature, this book is based on well-known algorithms of which several – those used in the examples – are described in detail. This book cannot, however, provide a detailed presentation of all the algorithms used in the STL. For example, readers who want to know more about red-black trees or quicksort should refer to other books about algorithms. The authors of the STL refer to Cormen *et al.* (1994) which is a very thorough book and well worth reading. An introduction to the STL is provided by Musser and Saini (1996), published while I was working on the first edition of this book. Josuttis (1999) describes the

C++ standard library including the STL part, but without the applications and the extensions presented in this book.

1.1 Genericity of components

An interesting approach is not to emphasize inheritance and polymorphism, but to provide containers and algorithms for all possible (including user-defined) data types, provided that they satisfy a few preconditions. C++ templates constitute the basis for this. Thus, the emphasis is not so much on object orientation but on generic programming. This has the very important advantage that the number of different container and algorithm types needed is drastically reduced – with concomitant type security.

Let us illustrate this with a brief example. Let us assume that we want to find an element of the `int` data type in a container of the `vector` type. For this, we need a `find()` algorithm which searches the container. If we have n different containers (list, set, ...), we need a separate algorithm for each container, which results in n `find()` algorithms. We may want to find not only an `int` object, but an object of an arbitrary data type out of m possible data types. This would raise the number of `find()` algorithms to $n \cdot m$. This observation will apply to k different algorithms, so that we have to write a total of $k \cdot n \cdot m$ algorithms.

The use of templates allows you to reduce the number m to 1. STL algorithms, however, do not work directly with containers but with interface objects, that is, iterators which access containers. Iterators are pointer-like objects which will be explained in detail later. This reduces the necessary total to $n + k$ instead of $n \cdot k$, a considerable saving.

An additional advantage is type security, since templates are already resolved at compile time.

1.2 Abstract and implicit data types

Abstract data types encapsulate data and functions that work with this data. The data is not visible to the user of an abstract data type, and access to data is exclusively carried out by functions, also called methods. Thus, the abstract data type is specified by the methods, not by the data. In C++, abstract data types are represented by classes which present a tiny flaw: the data that represents the state of an object of this abstract data type is visible (though not accessible) in the `private` part of the class declaration for each program that takes cognizance of this class via `#include`. From the standpoint of object orientation, 'hiding' the private data in an implementation file would be more elegant.

Implicit data types can on the one hand be abstract data types themselves, on the other hand they are used to implement abstract data types. In the latter case they are not visible from the outside, thus the name 'implicit.' For example: an abstract data type `Stack` allows depositing and removing of elements only from the 'top.' A

stack can, for instance, use a singly-linked list as implicit data type, though a vector would be possible as well. Users of the stack would not be able to tell the difference.

Implicit data types are not important in the sense of an object-oriented analysis which puts the emphasis on the interfaces (methods) of an abstract data type. They are, however, very important for design and implementation because they often determine the run time behavior. Frequently, a non-functional requirement, such as compliance with a given response time, can be fulfilled only through a clever choice of implicit data types and algorithms. A simple example is the access to a number of sorted addresses: access via a singly-linked list would be very slow compared to access via a binary tree.

The STL uses the distinction between abstract and implicit data types by allowing an additional choice between different implicit data types for the implementation of some abstract data types.

1.3 The fundamental concept

The most important elements of the STL are outlined before their interplay is discussed.

1.3.1 Containers

The STL provides different kinds of containers which are all formulated as template classes. Containers are objects which are used to manage other objects, where it is left to the user to decide whether the objects are deposited by value or by reference. 'By value' means that each element in the container is an object of a copyable type (value semantics). 'By reference' means that the elements in the container are pointers to objects of possibly heterogeneous type. In C++, the different types must be derived from a base class and the pointers must be of the 'pointer to base class' type.

A means of making different algorithms work with different containers is to choose the same *names* (which are evaluated at compile time) for similar operations. The method `size()`, for example, returns the number of elements in a container, no matter whether it is of `vector`, `list`, or `map` type. Other examples are the methods `begin()` and `end()` which are used to determine the position of the first element and the position *after* the last element. These positions are always defined in a C++ array. An empty container is characterized by identical values of `begin()` and `end()`.

1.3.2 Iterators

Iterators work like pointers. Depending on the application, they can be common pointers or objects with pointer-like properties. Iterators are used to access container elements. They can move from one element to the other, with the kind of movement being hidden to the outside (control abstraction). In a vector, for example, the `++`

operation means a simple switch to the next memory position, whereas the same operation in a binary search tree is associated with a traversal of the tree. The different iterators will be described in detail later.

1.3.3 Algorithms

The template algorithms work with iterators that access containers. Since not only user-defined data types, but also the data types already existing in C++, such as `int`, `char`, and so on are supported, the algorithms have been designed in such a way that they can also work with normal pointers (see the example in the following section).

1.3.4 Interplay

Containers make iterators available, algorithms use them:

Containers $\Longleftrightarrow$ Iterators $\Longleftrightarrow$ Algorithms

This leads to a separation which allows an exceptionally clear design. In the following example, variations of one program will be used to show that algorithms function just as well with C arrays as with template classes of the STL.

In this example, an `int` value to be entered in a dialog is to be found in an array, by using a `find()` function which is also present as an STL algorithm. In parallel, `find()` is formulated in different ways in order to visualize the processes. The required formulation is approached step by step by presenting a variation *without* usage of the STL. The container is a simple C array. To show that a pointer works as an iterator, the type name `IteratorType` is introduced with `typedef`.

```
// k1/a3.4/main.cpp
// variation 1, without using the STL
#include<iostream> // see Section 1.7.2 for header conventions
using namespace std;

// new type name IteratorType for pointer to const int
// (we don't want to modify the values here)
typedef const int* IteratorType;

// prototype of the algorithm
IteratorType find(IteratorType begin, IteratorType end,
                  const int& Value);

int main() {
    const int Count = 100;
    int aContainer[Count];           // define container

    IteratorType begin = aContainer;     // pointer to the beginning

    // position after the last element
    IteratorType end = aContainer + Count;
```

```
    // fill container with even numbers
    for(int i = 0; i < Count; ++i)
       aContainer[i] = 2*i;

    int Number = 0;
    while(Number != -1) {
       cout << " enter required number (-1 = end):";
       cin >> Number;
       if(Number != -1) {            // continue?
         IteratorType position = find(begin, end, Number);

         if (position != end)
            cout << "found at position "
                      << (position - begin) << endl;
         else
            cout << Number << " not found!" << endl;
       }
    }
}

// implementation
IteratorType find(IteratorType begin, IteratorType end,
              const int& Value) {
    while(begin != end        // pointer comparison
          && *begin != Value) // dereferencing and object comparison
         ++begin;             // next position
    return begin;
}
```

It can be seen that the `find()` algorithm itself does not need to know anything about containers. It only uses pointers (iterators) which need to have very few capabilities:

- The `++` operator is used to proceed to the next position.
- The `*` operator is used for dereferencing. Applied to a pointer (iterator), it returns a reference to the underlying object.
- The pointers must allow comparison by means of the `!=` operator.

The objects in the container are compared by means of the `!=` operator. In the next step, we cancel the implementation of the `find()` function and replace the prototype with a template:

```
// variation 2: algorithm as template (see k1/a3.4/maint1.cpp)
template<class Iteratortype, class T>
Iteratortype find(Iteratortype begin, Iteratortype end,
                  const T& Value) {
    while(begin != end        // iterator comparison
```

```
          && *begin != Value)  // dereferencing and object comparison
         ++begin;              // next position
    return begin;
}
```

The rest of the program remains *unchanged*. The placeholder `IteratorType` for the iterator's data type may have an arbitrary name. In the third step, we use a container of the STL. The iterators `begin` and `end` are replaced with the methods of the `vector<T>` class which return a corresponding iterator.

```
// variation 3 : a container as STL template (see k1/a3.4/maint2.cpp)
#include<iostream>
#include<vector>  // STL
using namespace std;

// new type name IteratorType for reading purposes, maybe (or
// maybe not!) equal to 'pointer to const int'
// (depends on implementation)
typedef vector<int>::const_iterator IteratorType;

// algorithm as template
template<class Iteratortype, class T>
Iteratortype find(Iteratortype begin, Iteratortype end,
                  const T& Value) {
    while(begin != end              // iterator comparison
          && *begin != Value)       // object comparison
         ++begin;                   // next position
    return begin;
}

int main() {
    const int Count = 100;
    vector<int> aContainer(Count);    // define container
    for(int i = 0; i < Count; ++i)    // fill container with
        aContainer[i] = 2*i;          // even numbers

    int Number = 0;
    while(Number != -1) {
       cout << " enter required number (-1 = end):";
       cin >> Number;
       if(Number != -1)   {
         // use global find() defined above
         IteratorType position =
            ::find(aContainer.begin(), // use of container methods:
                 aContainer.end(), Number);

         if (position != aContainer.end())
            cout << "found at position "
```

```
                    << (position - aContainer.begin()) << endl;
            else cout << Number << " not found!" << endl;
        }
    }
}
```

This shows how the STL container cooperates with our algorithm and how arithmetic with iterators is possible (formation of a difference). In the last step we use the `find()` algorithm contained in the STL and replace the whole template with an additional `#include` instruction:

```
// variation 4: STL algorithm (k1/a3.4/maintstl.cpp)
#include<algorithm>
// ... the rest as variation 3, but without find() template. Also the call ::find()
// has to be replaced with find() (i.e. std::find()).
```

In addition to this, it is not necessary to define an iterator type with `typedef` because every container of the STL supplies a corresponding type. So instead of `IteratorType`, you may write `vector<int>::iterator` in the above program. An interesting fact is that the algorithm can cooperate with *any* class of iterators that provides the operations `!=` for comparison, `*` for dereferencing, and `++` for proceeding to the next element. This is one reason for the power of the concept and for the fact that each algorithm has to be present in only *one* form, which minimizes management problems and avoids inconsistencies. Thus, algorithms and containers of the STL come quite close to the ideal concept that one can simply plug together various software components which will then function with each other.

The use of the large number of algorithms and containers of the STL makes programs not only shorter, but also more reliable, because programming errors are prevented. This helps to increase productivity in software development.

1.4 Internal functioning

How does the STL function internally? To show this in detail, the example from the previous section will be used, not with a container of the STL, but with a user-defined class which behaves exactly as the classes of the STL. To ensure that an iterator of this class cannot simply be identified with a pointer, the example must be made slightly more complex: instead of the vector, we take a singly-linked list. The class will be called `slist` (for simple list).

Thus, we have no random access to the elements via the index operator. Therefore, the container is filled by means of the method `push_front()`. Furthermore, to keep the class as simple as possible, no run time optimization is considered. This class for a simple list is not complete; it provides only what is needed in the example.

The predefined `find()` algorithm is used to show that the user-defined class really behaves exactly like a class of the STL.

The list consists of list elements whose type is defined inside the `list` class as a nested public class (`struct`). In a structure, direct access to internal data is possible,

but this is not a problem here because the data is located in the private section of the `slist` class. Each list element carries the pertinent data, for example a number, together with a pointer to the next list element. `firstElement` is the pointer to the first list element. The class `slist` provides an iterator type `iterator` which is located in the `public` section since it is to be publicly accessible. It is also used in the following `main()` program. An iterator object stores the current container position in the `current` attribute. The methods satisfy the requirements for iterators formulated on page 7.

```
// file k1/a4/slist.h : list template for singly-linked lists
// T is a placeholder for the data type of a list element
// incomplete! (only functions needed for the example are implemented)
#ifndef SIMPLELIST_H
#define SIMPLELIST_H

namespace br_stl {

#include<cassert>
#include<iterator>

template<class T>
class slist {
  public:

    /*Some types of the class get public names. Then it is possible to use them outside
      the class without knowing the implementation.
    */
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
    // etc. see text

    slist() : firstElement(0), Count(0) {}

    /*copy constructor, destructor and assignment operator are omitted! The implemen-
      tation of push_front() creates a new list element and inserts it at the beginning
      of the list:
    */
    void push_front(const T& Datum) { // insert at beginning
      firstElement =  new ListElement(Datum, firstElement);
      ++Count;
    }

  private:
    struct ListElement {
```

```
      T Data;
      ListElement *Next;
      ListElement(const T& Datum, ListElement* p)
      : Data(Datum), Next(p) {}
  };

  ListElement *firstElement;
  size_t Count;

public:
  class iterator {
     public:
      typedef std::forward_iterator_tag iterator_category;
      typedef T value_type;
      typedef T* pointer;
      typedef T& reference;
      typedef size_t size_type;
      typedef ptrdiff_t difference_type;

       iterator(ListElement* Init = 0)
       : current(Init){}

       T& operator*() {              // dereferencing
           return current->Data;
       }
       const T& operator*() const { // dereferencing
           return current->Data;
       }
       iterator& operator++() {      // prefix
          if(current) // not yet arrived at the end?
             current = current->Next;
          return *this;
       }
       iterator operator++(int) {    // postfix
          iterator temp = *this;
          ++*this;
          return temp;
       }
       bool operator==(const iterator& x) const {
          return current == x.current;
       }
       bool operator!=(const iterator& x) const {
          return current != x.current;
       }
```

```
    private:
      ListElement* current; // pointer to current element
  }; // iterator

  /*As can be seen above, in the postfix variation of the ++ operator, the copy constructor is needed for initialization and return of temp. For this reason, the prefix variation should be preferred where possible. Some methods of the slist class use the iterator class:
  */
  iterator begin() const { return iterator(firstElement);}
  iterator end()   const { return iterator();}
};
} // namespace br_stl
#endif // SIMPLELIST_H
```

Sometimes it is advantageous to write

```
// internal type definition may be unknown
slist<myDataType>::difference_type Dist;
```

in a program instead of

```
// predefined type
long Dist;
```

This is especially useful if there are possible later changes in the internal type structure of class `slist`. Using the public type names avoids changing an application program which uses the list. More advantages of exporting types will be described in Section 2.1.

At this point we only need the subtraction operator to be able to calculate differences between list iterators.

```
// (insert in slist.h above)
template<class Iterator>
Iterator::difference_type operator-(Iterator second,
                                    Iterator first) {
  Iterator::difference_type count = 0;  // type maybe int
  /*The difference between the iterators is determined by incrementing first until the second iterator is reached. Thus, the condition is that first lies not after the second iterator. In other words: second must be able to reach the iterator by means of the ++ operator.
  */
  while(first != second
        && first != Iterator()) {
    ++first;
    ++count;
  }

  // In case of inequality, second is not reachable by first
  assert(first == second);
```

```
  return count;
}
```

The loop condition involving `iterator()` (together with the assertion) ensures that the loop does not run endlessly and that the program aborts when the iterator cannot be reached from the iterator `first` by means of the `++` operation.

The following `main()` program strongly resembles the one on page 8 and uses the user-defined class in the same way as a class of the STL. Try using this example to get a clear idea of the functioning details, and you won't have any great problem understanding the STL.

```
// k1/a4/mainstl2.cpp
#include<algorithm>                  // contains find()
#include<iostream>
#include"slist.h"                    // user-defined list class (see above)

int main() {
    const int count = 100;
    br_stl::slist<int> aContainer;           // define the container
    /*Change of order because the container is filled from the front! This example differs
      from those in Section 1.3.4, because elements are inserted, i.e., the container is
      expanded as needed.
    */

    for(int i = count; i >= 0; --i)  {   // fill the container with
       aContainer.push_front(2*i);       // even numbers
    }

    int Number = 0;
    while(Number != -1) {
       std::cout << " enter required number (-1 = end):";
       std::cin >> Number;

       if(Number != -1) {
          // use of container methods:
         br_stl::slist<int>::iterator Position =
             std::find(aContainer.begin(),
                aContainer.end(), Number);

         if(Position != aContainer.end())
            std::cout << "found at position "
                      << (Position - aContainer.begin())
                      << std::endl;
         else
            std::cout << Number << " not found!"
                      << std::endl;
       }
    }
}
```

Exercise

1.1 Complete the `slist` class using the following:

- A method `iterator erase(iterator p)` that removes the element pointed to by the iterator `p` from the list. The returned iterator is to point to the element following `p` provided it exists. Otherwise, `end()` is to be returned.
- A method `void clear()` that deletes the whole list.
- A method `bool empty()` that shows whether the list is empty.
- A method `size_t size()` that returns the number of elements.
- A copy constructor and an assignment operator. The latter can utilize the first: Create a temporary copy of the slist and then exchange the management information (attributes).
- A destructor.

1.5 Complexity

The STL has been developed with the aim of achieving high efficiency. Run time costs are specified for each algorithm depending on the size and kind of the container to be processed. The only assumption made is that user-defined iterators can move from one element of a container to the next element in constant time.

This section gives a brief introduction to the concept of complexity as a measure for computing and memory requirements.

An algorithm should obviously be correct – this is, however, not the only requirement. Computer resources are limited. Thus, another requirement is that algorithms must be executed on a real machine in a finite number of cycles. The main resources are computer memory and available computing time.

Complexity is the term that describes the behavior of an algorithm with regard to memory and time consumption. The efficiency of an algorithm in the form of a running program depends on:

- the hardware,
- the type and speed of required operations,
- the programming language, and
- the algorithm itself.

The concept of complexity exclusively concerns the algorithm. Machine properties and programming language details are ignored, since they modify the time needed for an algorithm by a constant factor if we assume a von Neumann architecture. There are two ways of analyzing the efficiency of an algorithm:

1. Measurements

   - Carry out measurements of the run time behavior for different sets of input data.
   - The best, worst, and average cases are of interest. The cases depend on the properties of the input data, the system environment and the algorithm, so that corresponding knowledge must be available.

2. Analysis of the algorithm

   - The algorithm is analyzed. Machine, operating system and compiler are ignored.
   - The frequency of executed instructions is an index of the speed. This frequency can be directly derived from the algorithm.
   - Again, the best, worst, and average cases are of interest.

Only the second way will be described. Wherever the term 'complexity' appears, it is intended to mean *time complexity*. Examples can be found in Table 1.1. Since they are independent of any special programming language, they are written in pseudo-code notation. The symbol $\propto$ stands for 'proportional to.'

The loop variables i and j are of no importance in this context. The frequencies with which the instructions $x = x + y$ and $n = n/2$ in Table 1.1 are executed differ by *orders of magnitude* which are not dependent on any machine or programming language. Only these orders of magnitude are of interest here.

1.5.1 O notation

The 'O notation' describes an order of magnitude. In the examples of Table 1.1, the orders of magnitude $O(1)$, $O(n)$, $O(n^2)$, and $O(\log n)$ occur. Apart from a constant factor, the 'O notation' describes the maximum execution time for large values of n, thus indicating an *upper bound*. What 'large' means depends on the individual case, as will be shown in one of the following examples. The constant factor comprises all environmental properties in which the algorithm runs, like CPU speed, compiler etc. Ignoring the constant factor allows for comparing algorithms.

Definition: Let $f(n)$ be the execution time of an algorithm. This algorithm is of (time) complexity $O(g(n))$ if and only if two positive constants c and n_0 exist so that $f(n) \leq cg(n)$ applies to all $n \geq n_0$.

Example

Let us assume an algorithm for vectors whose execution time $f(n)$ depends on the length n of the vector. Let us further assume that

$$f(n) = n^2 + 5n + 100$$

Algorithm	Frequency	(Time) complexity
$x = x + y$	1	constant
for $i = 1$ to n *do* $x = x + y$ *od*	$\propto n$	linear

for $i = 1$ to n
do
  for $j = 1$ to n
  do $x = x + y$
  od
od
	$\propto n^2$	quadratic

$n =$natural number
$k = 0$
while $n > 0$
do
  $n = n/2$
  $k = k + 1$
od
	$\propto \log n$	logarithmic

Table 1.1: Algorithms and frequency.

applies. The execution time could now be estimated with a simpler function $g(n) = 1.1n^2$. If we now compare $f(n)$ with $g(n)$, we see that $g(n) > f(n)$ for all $n \geq 66$. Obviously, we could have chosen different values for c and n_0, for example $c = 1.01$ and $n_0 = 519$. Therefore, complexity of $f(n)$ is $O(n^2)$. The complexity says *nothing* about *actual* computing time.

Example

Let A be an algorithm of execution time $f_A(n) = 10^4 n$ and B be an algorithm of execution time $f_B(n) = n^2$. It can easily be seen that algorithm A is faster for all values $n > 10^4$, whereas B is faster for all $n < 10^4$. Algorithm A is therefore to be recommended for large values of n, where in this case, the word 'large' means $n > 10^4$.

Therefore, algorithms of low complexity should normally be preferred. Exceptions are possible, depending on the value of the constants c and n_0. In order to select an appropriate algorithm for a given problem, the size n of the input data set must be known.

Some rules

1. $O(const * f) = O(f)$ Example: $O(2n) = O(n)$

		Examples:
2.	$O(f * g) = O(f) * O(g)$	$O((17n) * n) = O(17n) * O(n)$ $= O(n) * O(n) = O(n^2)$
	$O(f/g) = O(f) * O(\frac{1}{g})$	$O((3n^3)/n) = O(3n^2) = O(n^2)$
3.	$O(f + g)$ = dominating function of $O(f)$ and $O(g)$	$O(n^5 + n^2) = O(n^5)$

Examples

Linear search
Let us assume an unordered sequence of names together with addresses and phone numbers. The task is to find the phone number for a given name.

- The number to be found can lie at the beginning, the end, or somewhere in the middle.
- On average, we must compare $n/2$ names when the total number of names is n.
- The time complexity is $O(n/2) = O(n)$.

Binary search
Now, we look for a name in a normal, thus sorted, phone book.

- We look in the middle of the book and find a name. If this is the name we are looking for, we have finished. If not, we continue our search in the left or right half of the book, depending on whether the name we are looking for is alphabetically located before or after the name we just saw.
- We repeat the previous step with the chosen half of the book until we have found the name we are looking for, or we find out that it does not occur in the book at all. With each of these steps, the number of possible names is halved: $n/2, n/4, n/8, ..., 4, 2, 1$.
- There exists a number k so that $n \geq 2^{k-1}$ and $n \leq 2^k$. We do not need more than k comparisons.
- The algorithm is of complexity $O(k) = O(\log_2 n) = O(\log n)$.

Travelling salesman problem (TSP)
A travelling salesman wants to visit n towns. He wants to save time and money and looks for the shortest route that connects all towns. One method to find the optimum solution is an analysis of all possible routes. What is the complexity of this method?

As his first town, he can choose one out of n towns. From this point, he can choose between $n-1$ towns to drive to next. When he has reached the next town, he can choose between $n-2$ towns, and so on. When he has visited $n-1$ towns, only one choice remains: town number n. The total number of routes to connect n towns is $n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1 = n!$

If 20 towns are to be visited, there are 20! = 2,432,902,008,176,640,000 different routes to compare. The complexity of the algorithm is $O(n!)$.

This well-known problem is an example of a class of similar problems which are called NP complete. NP is an abbreviation for 'non-deterministic polynomial.' This means that a non-deterministic algorithm (which 'magically' knows which is to be the next step) can solve the problem in polynomial time ($O(n^k)$). (A more extensive and more serious treatment of the subject can be found in Hopcroft and Ullman (1979).) In the end, it does not matter at all *in which order* the next town to be visited is chosen, but if you *do* know the right order, the solution is found very quickly.

However, predefining an order changes the algorithm into a deterministic one, and because magic does not work, we usually have no choice other than predefining a schematic order – and there we are! Only occasionally does experience help with specially structured problems. As far as the salesman is concerned, this means that there is no deterministic algorithm with a polynomial time function n^c (c = constant) that dominates $n!$ For each possible constant c there exists an n_0, so that for all n greater than n_0, $n!$ is greater than n^c.

The class of NP problems is also called 'intractable,' because for a large number of input variables, solution attempts do not arrive at a result in a reasonable time measured on a human timescale. On the other hand, existing solutions of NP problems can be verified 'quickly,' that is, in polynomial time.

A mathematical proof that the salesman problem and other related problems can have no polynomial solution is still pending. There are some heuristic methods which at least approach the optimum and are significantly faster than $O(n!)$.

This class of problems has practical applications, for example:

- drilling hundreds or thousands of holes in a circuit board with a moving laser in a minimum time,
- finding the cheapest path in a computer network,
- distributing goods in a region using a shipping agency.

1.5.2 Ω notation

The O notation defines an *upper* bound for an algorithm. Improvement of an algorithm can reduce the bound. For example, sequential search in a sorted table: $O(n)$, binary search in a sorted table: $O(\log n)$. Is there also a *lower* bound for a given algorithm? Is it possible to show that the solution of a given problem requires a certain minimum of effort?

If a problem necessitates *at least* $O(n^2)$ steps, there is no point in searching for an $O(n)$ solution.

The Ω notation describes lower bounds. For example, sequential search in a table is of the order $\Omega(n)$, because each element must be looked at at least once. $\Omega(\log n)$ is not possible. In this case, $\Omega(n) = O(n)$.

Example

Multiplication of two $n * n$ matrices:

upper bound:	
$O(n^3)$	simple algorithm (three nested loops)
$O(n^{2.81})$	von Strassen 1969
$O(n^{2.376})$	Coppersmith and Winograd 1987 (quoted in Cormen *et al.* (1994))
lower bound:	
$\Omega(n^2)$	at least two loops are needed, because n^2 elements must be entered into the product matrix

1.6 Auxiliary classes and functions

This section briefly describes some tools which will be needed at a later stage.

1.6.1 Pairs

A *pair* in the sense of the STL is an encapsulation of two objects which belong together and which can be of different types. Pairs are fundamental components which will be used in later chapters. They are defined by means of a public (`struct`) template class, defined in header `<utility>`:

```
template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair(){}; // see text
    // initialize first with x and second with y:
    pair(const T1& x, const T2& y);
    // copy constructor:
    template<class U, class V> pair(const pair<U, V> &p);
};
```

The default constructor causes the elements to be initiated with the default constructors of their type. In addition to the class definition, there are some comparison operators:

```
template <class T1, class T2>
bool operator==(const pair<T1, T2>& x,
                const pair<T1, T2>& y) {
    return x.first == y.first && x.second == y.second;
}
```

```
template <class T1, class T2>
bool operator<(const pair<T1, T2>& x,
               const pair<T1, T2>& y) {
    return    x.first < y.first
           || (!(y.first < x.first)
           && x.second < y.second);
}
```

When the first objects are equal, the return value of the < operator is determined by the comparison of the second objects. However, in order to make only minimum demands on the objects, the equality operator == is not used in the second template. It might be the case that equality of two pairs is not required in a program. Then, the above template `operator==()` is not applied, so that the classes `T1` and `T2` only have to provide the < operator. The other comparison operators for pairs are

```
template <class T1, class T2>
  bool operator!=(const pair<T1, T2>& x,
                  const pair<T1, T2>& y);
template <class T1, class T2>
  bool operator> (const pair<T1, T2>& x,
                  const pair<T1, T2>& y);
template <class T1, class T2>
  bool operator>=(const pair<T1, T2>& x,
                  const pair<T1, T2>& y);
template <class T1, class T2>
  bool operator<=(const pair<T1, T2>& x,
                  const pair<T1, T2>& y);
```

A function facilitates the generation of pairs:

```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y) {
    return pair<T1, T2>(x, y);
}
```

`pair` objects are needed from Section 4.4.1 onward.

1.6.2 Comparison operators

In namespace `std::rel_ops`, the STL provides comparison operators which make it possible that in a class only the operators == and < must be defined and yet the whole set of comparisons is available:

```
template <class T>
bool operator!=(const T& x, const T& y) {
```

```
    return !(x == y);
}

template <class T>
bool operator>(const T& x, const T& y) {
    return y < x;
}

template <class T>
bool operator<=(const T& x, const T& y) {
    return !(y < x);
}

template <class T>
bool operator>=(const T& x, const T& y) {
    return !(x < y);
}
```

Strictly speaking, it would be possible to manage with only the < operator if the following definition is contained in the STL:

```
// not part of the STL!
template <class T>
bool operator==(const T& x, const T& y) {
    return !(x < y) && !(y < x);
}
```

This kind of check is sometimes used inside the STL. Strictly speaking, the term 'equality' is no longer appropriate; one should actually use the term 'equivalence.' When comparing integer numbers with the < operator, the two terms coincide; this is, however, not generally the case, as the following example shows. In Webster's International Dictionary, accented letters are treated in the same way as the corresponding simple vowels. Thus, 'pièce de résistance' stands between 'piece by piece' and 'piece-meal.' 'pièce' and 'piece' are not equal, but equivalent with respect to sorting. Another way of carrying out comparisons is shown in Section 1.6.3.

1.6.3 Function objects

In an expression, the call of a function is replaced with the result returned by the function. The task of the function can be taken over by an object – a technique frequently employed in the algorithms of the STL. For this purpose, the function operator `()` is overloaded with the operator function `operator()()`.

Then an object can be called in the same way as a function. Algorithmic objects of this kind are called *function objects* or *functors*.

Functors are objects which behave like functions but have all the properties of objects. They can be generated, passed as arguments, or have their state modified. The change of state allows a flexible application which, with functions, would be only possible via additional parameters.

Comparisons

The STL provides a large number of template classes for comparisons. Objects of this class appear later under the name of comparison object. Table 1.2 shows the calls of objects as function calls, that is, `X(x,y)` is identical to `X.operator()(x,y)`.

The comparison classes are binary functions, and therefore they inherit from the `binary_function` class. The only purpose of this class is to provide uniform type names for all classes inheriting from it:

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

For unary classes, a corresponding template `unary_function` is defined. The word `struct` saves the `public` label. Everything can be public, because the class has no data to be protected. This, for example, is the `equal_to` template for equality:

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

Object definition (Type `T`)	Call	Return
`equal_to<T> X;`	`X(x, y)`	`x == y`
`not_equal_to<T> X;`	`X(x, y)`	`x != y`
`greater<T> X;`	`X(x, y)`	`x > y`
`less<T> X;`	`X(x, y)`	`x < y`
`greater_equal<T> X;`	`X(x, y)`	`x >= y`
`less_equal<T> X;`	`X(x, y)`	`x <= y`

Table 1.2: Template classes for comparison (header `<functional>`).

The aim of templates is to supply algorithms with a uniform interface. The templates rely on the corresponding operators of data type `T`. However, a specialized comparison class can be written for user-defined classes without having to change the algorithm. The user-defined class does not even need to have the comparison operators ==, < and so on. This technique is used quite frequently; at this point, a short example will demonstrate how it functions.

A normal C array of `int` numbers is sorted once by element size using the standard comparison object `less<int>` and once by the *absolute value* of the elements, where in the second case, a user-defined comparison object `absoluteLess` is used. To show the effect more clearly, a normal C array and a modest function template

`bubble_sort` are used instead of accessing the containers and algorithms of the STL.

```
// k1/a6/compare.cpp – Demonstration of comparison objects
#include<functional>       // less<T>
#include<iostream>
#include<cstdlib>          // abs()

struct absoluteLess {
    bool operator()(int x, int y) const {
       return abs(x) < abs(y);
    }
};
```

The following sorting routine no longer uses the < operator in the `if` condition, but the comparison object whose `operator()(...)` is called. Please ignore the bad performance, later we'll see the much faster `sort()` (see page 122).

```
template<class T, class CompareType>
void bubble_sort(T* array, int Count,
                 const CompareType& Compare) {
    for(int i = 0; i < Count; ++i) {
        for(int j = i+1; j < Count; ++j)
        if (Compare(array[i], array[j])) { // functor call
           // exchange
           const T temp = array[i];
           array[i] = array[j];
           array[j] = temp;
        }
    }
}

// Auxiliary procedure for display
void Display(int *Array, int N) {
  for(int i = 0; i < N; ++i) {
       std::cout.width(7);
       std::cout << Array[i];
  }
  std::cout << std::endl;
}

int main() {
    int Table[] = {55, -7, 3, -9, 2, -9198, -937, 1, 473};
    const int num = sizeof(Table)/sizeof(int);

    /*The comparison object normalCompare is of the standard class type less,
      which has been made known with #include<functional>. less compares
      with the < operator..
    */
```

```
    // Variation 1
    std::less<int> normalCompare;
    bubble_sort(Table, num, normalCompare);
    std::cout << "sorted by size:" << std::endl;
    Display(Table, num);
        // 473 55 3 2 1 –7 –9 –937 –9198

    /*Alternatively, you can do without explicit creation of an object when the construc-
      tor is called in the argument list.
    */
    // Variation 2
    bubble_sort(Table, num, std::less<int>());

    /*The comparison object is of the user-defined type absoluteLess which uses
      not only the < operator, but also the abs() function, and which in principle can
      be arbitrarily complex. It is a big advantage that the bubble_sort algorithm
      and its interface do not have to be changed.
    */

    std::cout << "sorted by absolute value:" << std::endl;
    bubble_sort(Table, num, absoluteLess());
    Display(Table, num);
        // –9198 –937 473 55 –9 –7 3 2 1
}   // End of example
```

The user-defined design of special comparison functions shows the great flexibility of the concept of function objects. In addition to the examples shown, appropriately written function objects can also carry data, if needed.

Arithmetic and logic operations

As in the previous section, the STL provides template classes for arithmetic and logic operations (see Table 1.3) which can be used like a function by means of the overloaded `operator()`. (Note that 'multiplies' was called 'times' in earlier draft versions of the C++ standard.) The advantage is again that these templates can be overloaded with specializations without having to change the interfaces of the algorithms involved.

1.6.4 Function adapters

Function adapters are nothing more than function objects which cooperate with other function objects to adapt them to different requirements. This allows us to get by with existing functors and avoid writing new ones.

not1

The function `not1` takes a functor as the parameter which represents a predicate with *one* argument (thus the suffix 1) and returns a functor which converts the

Object definition (Type T)	Call	Return
plus<T> X;	X(x, y)	x + y
minus<T> X;	X(x, y)	x - y
multiplies<T> X;	X(x, y)	x * y
divides<T> X;	X(x, y)	x / y
modulus<T> X;	X(x, y)	x % y
negate<T> X;	X(x)	-x
logical_and<T> X;	X(x, y)	x && y
logical_or<T> X;	X(x, y)	x || y
logical_not<T> X;	X(x)	!x

Table 1.3: Arithmetic and logic template classes (header `<functional>`*).*

logical result of the predicate into its opposite. Let us assume that there exists a predicate `odd` with the following definition (that by the way, can be replaced with `bind2nd(modulus<int>(),2)`, see page 26):

```
struct odd : public unary_function<int, bool> {
    bool operator () (int x) const {
      return (x % 2) != 0;
    }
};
```

Application of `not1` is shown by the following program fragment:

```
int i;
cin >> i;
if(odd()(i))
   cout << i << " is odd";
if(not1(odd())(i))
   cout << i << " is even";
```

Instead of an object declared on purpose, first a temporary object of type `odd` is generated whose operator `()` is called. In the second `if` instruction, `not1` generates a functor whose operator `()` is called with the argument `i`. How does this work? The STL provides a class out of which `not1` generates an object:

```
template <class Predicate>
class unary_negate
 : public unary_function<typename
                         Predicate::argument_type, bool> {
  protected:
    Predicate pred;
  public:
    explicit unary_negate(const Predicate& x) : pred(x) {}
    bool operator()(const typename
                    Predicate::argument_type& x) const {
```

```
        return !pred(x);
    }
};
```

The operator `()` returns the negated predicate. The class inherits the type definition of `argument_type` from `unary_function`. However, the compiler shall be able to identify the parameter type of `operator()()` without analyzing the `Predicate`-template. This is required by ISO/IEC (1998). Therefore `typename` is used.

not2

This function works in a similar way, but it refers to predicates with *two* parameters. This can be used to reverse the sorting order of variation 2 on page 24:

```
// Variation 2, reverse sorting order
bubble_sort(Table, num, std::not2(std::less<int>()));
```

Analogous to `not1`, internally a class `binary_negate` is used. The sorting order by *absolute value* on page 24 can be reversed with `not2` only if the class inherits from `binary_function` for comparisons (see page 22):

```
struct absoluteLess
  : public binary_function<int, int, bool> {
    ...  // as above
};
```

bind1st, bind2nd

These functions transform binary function objects into unary function objects by binding one of the two arguments to a value. They accept a function object with two arguments and a value `x`. They return a unary function object whose first or second argument is bound to the value `x`. For example, the known functor `less` (see Table 1.2) compares two values and returns true if the first value is less than the second one. If the second value is fixed, for example to 1000, a unary function object suffices which is generated by means of `binder2nd`. The `find()` algorithm described on page 6 has an overloaded variation described later (page 89) which accepts one predicate.

```
std::find(v.begin(), v.end(),
          std::bind2nd(std::less<int>(), 1000));
```

finds the first number in the `int` vector `v` which is less than 1000, and

```
std::find(v.begin(), v.end(),
          std::bind1st(std::less<int>(), 1000));
```

finds the first number in the `int` vector `v` which is greater than 1000. The functors returned by the functions `bind1st<operation, value>()` and `bind2nd<operation, value>()` are of the type `binder1st<operation, value>` and

`binder2nd <operation, value>`. In an application such as the one above, the types usually do not appear explicitly (class definition in header `<functional>`).

ptr_fun

This overloaded function transforms a pointer to a function into a functor. As an argument, it has a pointer to the function which can have one or two parameters. The function returns a function object which can be called in the same way as the function. The types of function objects defined in `<functional>` are

```
pointer_to_unary_function<parameter1, result>
```

and

```
pointer_to_binary_function<parameter1, parameter2, result>
```

Frequently (but not always), these types remain hidden in the application. A short example shows its use. A pointer to a function is initialized with the sine function. Subsequently, the sine of an angle is called both via the function pointer and via a function object generated with `ptr_fun()`.

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

double (*f)(double) = std::sin;          // initialize pointer

int main() {
   double alpha = 0.7854;
                                         // call as:
   std::cout << f(alpha)                 // function
        << std::endl
        << std::ptr_fun(f)(alpha)        // functor
        << std::endl;
}
```

1.7 Some conventions

1.7.1 Namespaces

In order to avoid any name clashes, nearly all sample classes in the book are in namespace `br_stl`.

In files with sample `main()`-programs, often `using namespace std;` is used. All other programs use qualified names like in `std::cout << std::endl;` instead of `cout << endl;`.

All algorithms and classes of the C++ standard library are in namespace `std`, even if this is not specially mentioned.

1.7.2 Header files

The standard calling conventions put all C-headers into the namespace `std`. For example, the standard header `<cctype>` is in the namespace `std`, whereas `<ctype.h>` is in the global namespace.

The C standard library functions are accessed by omitting the '.h' extension of the file name and prefixing the old file name with a 'c.' For example:

```
#include<string>      // C++ string class

#include<cstring>     // C string functions for C++, namespace std
#include<string.h>    // C string functions, global namespace

#include<cctype>      // ctype functions for C++
#include<ctype.h>     // ctype functions, global namespace
```

The sample programs available via the internet (see page 271) contain a special include-directory which should be passed to the compiler with the -I option. Therefore header files of this directory are included using angle brackets `<>` instead of quotation marks `"`. Some people prefer quotation marks. However, this means that the compiler first tries to look up headers in the current directory. Telling the compiler with the -I option where the header files really are saves compilation time.

1.7.3 Allocators

Allocators provide memory for containers. There are system provided standard allocators, but you can define your own special allocators which, for example, do some garbage collection. Allocators are not treated in this book, because the emphasis lies on data structures and algorithms and their complexity.