Computing in C++
Part II : **O**bject **O**riented **P**rogramming in C++

Dr Robert Nürnberg

What is OOP?

Object oriented programming is a fairly new way to approach the task of programming. It supersedes the so called *procedural* or *structured programming languages* like Algol, Pascal or C, that have been around since the 1960s. The essence of structured programming is to reduce a program into smaller parts and then code these elements more or less independently from each other.
Although structured programming has yielded excellent results when applied to moderately complex programs, it fails when a program reaches a certain size. To allow for more complex programs to be written, the new approach of OOP was invented. OOP, while allowing you to use the best ideas from structured programming, encourages you to decompose a problem into related subgroups, where each subgroup becomes a self-contained object that contains its own instructions and data that relate to that object. In this way complexity is reduced, reusability is increased and the programmer can manage larger programs more easily.
All OOP languages, including C++, share the following three capabilities:

- *Encapsulation* – the ability to define a new type and a set of operations on that type, without revealing the representation of the type.
- *Inheritance* – the ability to create new types that inherit properties from existing types.
- *Polymorphism* – the ability to use one name for two or more related but technically different purposes; "one interface, multiple methods."

Recommended books and sites

- Daniel Duffy, *Introduction to C++ for Financial Engineers : An Object-oriented Approach*, 2006
- Steve Oualline, *Practical C++ Programming*, 1995
- Herbert Schildt, *Teach yourself C++*, 1992
- Jesse Liberty, *Teach yourself C++ in 24 hours*, 1999
- Bruce Eckel, *Thinking in C++*, Online at `www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`
- Bjarne Stroustrup, *The C++ Programming Language*, 1997 (technical)
- `www.research.att.com/~bs/C++.html` (Stroustrup: C++)
- `www.cplusplus.com`
- `www.cppreference.com`

Compilers

- Windows: · Microsoft Visual C++ .NET with Integrated Development Environment (IDE)
  · GNU C++ compiler (`g++`) as part of `Cygwin` or `MinGW`, without IDE
  · Dev-C++ – free compiler/IDE that is GNU compatible
- Linux: · GNU C++ compiler (`g++`) – part of any distribution
  · Intel C++ compiler (`icc`) – free for students
- Mac: · Xcode – free compiler/IDE that is GNU compatible
- Windows/Linux/Mac: · Code::Blocks – free compiler/IDE that is GNU compatible
  · NetBeans – free compiler/IDE that is GNU compatible

Contents

1 Revision

Before we embark on the topic of object oriented programming, let's remind ourselves of some other important features of C++.

1.1 "Hello World!" revisited

In C++ this standard example should look like

```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

The first two lines of the above example show the new ANSI-C++ compliant way to include the standard C++ libraries. Although many compilers still support the old style, eventually all C++ compilers will only accept the ANSI standard.
Note that without the second line the code would have to look something like

```
#include <iostream>
using namespace std;

int main() {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

See §7.1 below for further details on namespaces.

1.2 Using C header files

All of the old C standard header files, like `stdlib.h`, `math.h`, etc. can still be used. This should be done as follows.

```
#include <iostream>
#include <cmath>                // was: math.h in C
using namespace std;

int main() {
  cout << "exp(1.0) = " << exp(1.0) << endl;
  return 0;
}
```

For more details on the ANSI-C++ way to include header files from the standard library visit `www.cplusplus.com/doc/ansi/hfiles.html`.

1.3 Macros

The C/C++ compiler evaluates certain preprocessing directives before actually starting to compile a source code. These preprocessing directives are lines in your program that start with `#`. An example is the `#include` statement, which tells the compiler what kind of system calls and libraries you want to use in your program.
A macro is another preprocessing directive. In its simplest form, a macro is just an abbreviation. It is a name which stands for a fragment of code. Usually, this will be some specific constant, such as `Pi` and `DEBUG` in the example below. Before you can use a macro, you must define it explicitly with the `#define` directive. It is followed by the name of macro and then the code it should be an abbreviation for.
Macros can also be defined to take arguments. In this case, any expansion you define for the macro, will be applied with the arguments substituted by the values you are giving. The function macros `SQR()` and `MAX()` in the example below demonstrate the usage.
Finally, you can also use conditional directives that allow a part of the program to be ignored during compilation, on some conditions. A conditional can test either an arithmetic expression or whether a

name is defined as a macro. Check the usage of `#if`, `#else`, `#endif`, and `#ifdef`, `#ifndef` in the example below.

```
#include <iostream>
using namespace std;

#define SQR(a) ((a)*(a))
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#ifndef Pi
  #define Pi 3.14159265358979323846264338327950288841971
#endif
#define DEBUG 1

int main() {

#ifdef MAX
  if (MAX(2.0,3.4) > 2.5) cout << "Square of pi is: " << SQR(Pi) << endl;
#endif

#if DEBUG
  cout << "Debugging switched on." << endl;
  cout << " --- " << __FILE__ << " line " << __LINE__ << "---  Compiled on "
       << __DATE__ << "." << endl;
#else
  cout << "Debugging switched off." << endl;
#endif
  return 0;
}
```

Note that the example also uses some standard predefined macros. They are available with the same meanings regardless of the machine or operating system on which you are using C/C++. Their names all start and end with double underscores, e.g. `__FILE__`, `__LINE__` and `__DATE__`. More information on the C/C++ preprocessor can be found at `doc.ddart.net/c/gcc/cpp_toc.html`.

1.4 Enumerations

As in C, enumerations provide a convenient way to define variables that have only a small set of meaningful values. Here is a simple example.

```
enum colour {red, green, blue, black};
colour foreground = black;
```

1.5 Strings

C++ may use the C string functions, where a string is represented by a `char*`, but they rely on a null termination and proper memory allocation to hold the string. The C++ string class attempts to simplify string manipulation by automating much of the memory allocation and management.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string a("abcd efg");
  string b = "xyz ijk";
  string c = a + b;                                    // concatenation
  cout << "First character of '"<< c <<"' is : " << c[0] << endl;    // as in C
  return 0;
}
```

1.6 Signatures of functions

In C++, in contrast to C, it is possible that two different functions share exactly the same name. That is because in C++ a function is identified by its *signature* which comprises both the function's name and

the list of arguments it takes. Hence the following code is perfectly fine.

```
#include <iostream>
using namespace std;

void add_some(int *i) { (*i) += 3; }
void add_some(double *x) { (*x) += 1.5; }

int main() {
  int a(1);
  double b(1.23);
  add_some(&a); add_some(&b);
  cout << "a = " << a << ", b = " << b << endl;    // a = 4, b = 2.73
  return 0;
}
```

This feature of C++ is also referred to as *function overloading*. More on that in §3.

1.7 Passing variables to functions

1.7.1 Reference vs. pointers

Passing a variable to a function by reference or by pointer is essentially the same thing. For instance, the following two functions do exactly the same.

```
#include <iostream>
using namespace std;

void add_one_pt(double *x) { ++(*x); }
void add_one_rf(double &x) { ++x; }

int main() {
  double a(1.23), b(0.0);
  add_one_pt(&a);
  add_one_rf(b);
  cout << "a = " << a << ", b = " << b << endl;
  return 0;
}
```

However, there are times when we have to use pointers in order to pass a variable to a function. In particular, if we cannot be sure that the referred to variable will always be defined, we have to pass it via a pointer. Here is a simple example of such an instance.

```
#include <iostream>
using namespace std;

void execute(void (*f)()) {
  if (f != NULL) f();
  else cout << "Nothing to execute!" << endl;
}

void print() {
  cout << "Hello World!" << endl;
}

int main() {
  void (*p)() = print;
  execute(NULL);
  execute(p);
  return 0;
}
```

1.7.2 Reference vs. value

When we do not want to change the value of a variable that is given as an argument to a function, we usually pass it to the function by value. This causes the function to create a local copy of the passed

variable. When passing large data structures this is usually undesirable. A way out is to pass the variable by reference using the keyword `const`. This ensures that the compiler will not allow any change to the passed variable within the function. The following two functions demonstrate the two concepts.

```
#include <iostream>
using namespace std;

void print_by_val(double x) { cout << x << endl; }
void print_by_ref(const double &x) { cout << x << endl; }

int main() {
  double p = 3.14;
  print_by_val(p);
  print_by_ref(p);
  return 0;
}
```

1.7.3 Keyword `const`

Similar to the above usage of `const` in connection with passing arguments by reference, it can be employed when passing arguments by pointers as well.

1.7.4 Default arguments

C++ offers the possibility to implement default arguments for functions, in case that they are called with less parameters than originally specified. Here is a simple example.

```
#include <iostream>
using namespace std;

void print(int i = 10, int j = 5) { cout << i << " " << j << endl; }

int main () {
  print(2);         // '2 5'
  return 0;
}
```

Note that all default parameters must be to the right of any parameters that do not have defaults. Also, once you define a default parameter, all the following parameters have to have a default as well.

1.8 File input and output

Input and output from and to a file needs the header file `fstream` and is similar to using `cin` and `cout`. E.g.

```
#include <fstream>
using namespace std;

int main() {
  ofstream fout("output.txt");
  fout << "Hello World!" << endl;
  fout.close();
  return 0;
}
```

When dealing with variable length input files, it should be noted that the insertion operator `>>` itself returns *true* whenever the last input operation has been successful. See the example below for an illustration.

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream fin("data.dat");
  if (! fin.is_open()) {
```

```
    cout << "Error opening the file data.dat." << endl;
    return -1;
  }
  double mean = 0.0, x;
  int n = 0;
  while ( fin >> x ) {
    mean += x;
    n++;
  }
  fin.close();
  if (n > 0) mean /= n;
  cout << "The mean of the " << n << " numbers is " << mean << "." << endl;
  return 0;
}
```

2 Objects

The term *object* in C++ refers to an instance of a class. In the example below, `studlist[0]` is an instance of the `struct student`, where we have noted that a `struct` (as known from C) is simply a class with special properties, see §2.2 for details.
The support for classes is the most significant difference between C and C++. A class is essentially an abstract data type. It is a collection of data together with functions that operate on the data. We will refer to these as *member data* and *member functions* of a class, respectively.
By declaring a class we can create a new data type that, if we are careful, is as powerful as any of the basic types.
We start our considerations with a simpler concept known from C.

2.1 Structures

Structures allow us to bundle data that naturally belong together into one object. E.g.

```
#include <iostream>
#include <string>
using namespace std;

struct student {
  string name, firstname;
  int year;
};

int main() {
  int N;
  cout << "Number of students: "; cin >> N;
  student *studlist = new student[N];
  for (int i = 0; i < N; i++) {
    cin >> studlist[i].name >> studlist[i].firstname >> studlist[i].year;
  }
  // etc
  return 0;
}
```

Note that a `struct` defines a *type* that can be used (almost) as freely as any of the basic types, such as `double`, `int`, `bool` etc. In particular, we can pass variables of this newly defined *type* to functions, use references and pointers to and declare arrays with these variables.

2.1.1 Pointers to structures

At times it may be preferable to use (arrays of) pointers to structures. In this case the `main()` program above would have to be changed to something like

```
int main() {
  int N;
  cout << "Number of students: "; cin >> N;
  student **studlist = new student* [N];
  for (int i = 0; i < N; i++) {
    studlist[i] = new student;
    cin >> studlist[i]->name >> studlist[i]->firstname >> studlist[i]->year;
  }
  // etc
  return 0;
}
```

Observe that here the members of a `struct` are accessed via e.g. `pointer->firstname`, whereas before we used `variable.firstname`.

2.2 Classes – private and public members

As mentioned earlier, classes and structures are almost the same. In fact, the only difference is that by default the members of a structure are accessible from the outside via the `.` and `->` operators. In a class, on the other hand, the members are by default hidden within the class and cannot be accessed from the outside. We say that the members of a class are declared *private* by default. However, if we want to, we can declare members of a class to be *public*.
Hence the `struct student` in §2.1 can equivalently be defined as

```
class student {
public:
  string name, firstname;
  int year;
};
```

Nevertheless, it is somewhat bad programming practice to declare *member data* to be public. For a safe and easy to update code one should always declare these as *private*.
But if the private members of a class are not accessible outside the class, we have to find a way to change and inspect their values. This can be done with the help of functions that have access to the member data and are themselves declared inside the class.

2.3 Methods

Member functions of a class are also referred to as *methods*. We have come across an instance of this already during file input and output, cf. §1.8, where the line `fout.close();` called the member function `close()` for the instance `fout` of the class `ofstream`.
If we make the member data of our student class private, we should provide functions to create and print the data, say. The declaration of the class would then look as follows.

```
class student {
  string name, firstname;
  int year;
public:
  void create(const string &n, const string &fn, int y);
  void print();
};
```

Observe that the member data is declared private, since this is the default setting for a class.
So far we have only declared the new member functions. Before we can actually use them, we have to give their definitions. This can be done as follows.

```
void student::create(const string &n, const string &fn, int y) {
  name = n; firstname = fn; year = y;
}
void student::print() {
  cout << name << ", " << firstname << " (YEAR " << year << ")" << endl;
}
```

Note the usage of the scope operator `::` in order to define the member functions.
Here is an example of how to use the newly defined methods.

```
int main() {
  student s1;
  s1.create("Smith", "Alan", 1);
  s1.print();
  return 0;
}
```

The big advantage of defining all the member data as private is that firstly we can easily trace an error if something goes wrong, since the only place where the data is changed is in some of the member functions. And secondly, it is very easy to enrich and modify the class later on. For example, we could choose to include a variable `address` in the class. All we have to do in order to make sure that this new variable is always properly initialized in our program is to change the `create` method to something like

```
void student::create(const string &n, const string &fn, int y) {
  name = n; firstname = fn; year = y; address = "unknown";
}
```

and maybe provide another `create` method to initialize all four variables, e.g.

```
void student::create(const string &n, const string &fn, int y, const string &a) {
  name = n; firstname = fn; year = y; address = a;
}
```

2.4 Constructors

In order to properly initialize an object, C++ allows to define a constructor function that is called each time an object of that class is created. A constructor function has the same name as the class of which it is a part and has no return type. On recalling default arguments from §1.7.4, the constructor for the original `student` class can be defined as follows.

```
class student {
  string name, firstname;
  int year;
public:
  student(const string &n = "unknown", const string &fn = "unkown", int y = 1) {
    name = n; firstname = fn; year = y;
  }
  void print();
};
```

2.4.1 Copy constructors

A constructor that takes as its only argument an object of its own class is called *copy constructor*. It is also used whenever a parameter is passed by value and when an object is returned by a function. If there is no copy constructor defined for the class, C++ uses the default built-in copy constructor, which creates a bit wise copy of the object. Hence one only needs to define this special constructor if the class has a pointer to dynamically allocated memory.
The declaration of the copy constructor in the case of the `student` class would have to look as follows.

```
class student {
public:
  student(const student &s);
  // etc
};
```

Observe that we have to pass the parameter by reference, since otherwise a copy of the argument would have to be made, which would invoke the copy constructor, for which another copy would have to be made ...etc. Note also that we used the keyword `const` to ensure that the argument is not changed inside the function.

2.5 Destructors

The complement of a constructor is a destructor. This function is called automatically whenever an object is destroyed, i.e. for local variables when they go out of scope and for global variables when the program ends. This special member function should never be called explicitly. A destructor needs to be

implemented only for classes that have a pointer to dynamically allocated memory, for all other classes the default destructor is sufficient. The destructor function has the name of the class preceded with a ~. It has no return type and takes no parameters.
Here is an example.

```
class array {
  double *a;
  int len;
public:
  array(int l = 0) { len = l; a = new double[len]; }        // constructor
  ~array() { delete[] a; }                                  // destructor
};
```

2.6 The `this` pointer

C++ contains a special pointer that is called `this`. `this` is a pointer that is automatically passed to any member function when it is called, and it is a pointer to the object that generates the call. E.g. given the statement `foo.bar()`, the function `bar()` is automatically passed a pointer to `foo`, which is the object that generates the call. This pointer is referred to as `this`.
Hence an equivalent definition of the function `student::print()` from §2.3 is

```
void student::print() {
  cout << this->name << ", " << this->firstname << " (YEAR " << this->year << ")" << endl;
}
```

Of course, there is no reason why one should use the `this` pointer as shown here, because the shorthand form is much easier. However, the `this` pointer has several uses, including aiding in overloading operators. See e.g. §3.4 for more details.

2.7 Keyword `const`

When passing objects by using the keyword `const`, as described in §1.7.3, as a parameter to a function that possibly calls some of the object's member functions, one has to tell C++ that these methods will not change the passed object. To do this one has to insert the keyword `const` in the methods' declarations. Here is a short example.

```
class myclass {
  int a;
public:
  void showvalues(const myclass &A) { this->print(); A.print(); }
  void print() const { cout << a << endl; }                    // const needed!
};
```

It is always advisable to use the keyword `const` for member functions that you as the programmer know will not change the object's data. For instance, it makes debugging easier and it helps to avoid some errors already at the compilation stage.

2.8 Classes inside classes

Apart from member data and member functions, classes can also define their own data types. These new class definitions, inside the given class, can then be used by the class's member functions as well as from outside the class. In the latter case, one has to use the scope operator. See the following example for details.

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  class local_type {
    double t;
  public:
    local_type(double x) { t = x; }
```

```
    void print() const { cout << "Local type: " << t << endl; }
  };
  void print() const { cout << a << endl; }
};

int main() {
  myclass::local_type z(2.3);
  z.print();
  return 0;
}
```

Note that the same access restriction for member data and member functions also applies to classes defined inside other classes. I.e. these classes can only be used from outside the class in which they are defined, if they were declared as `public`.

2.9 Friend functions

There will be times when the programmer wants a function to have access to the private members of a class without that function actually being a member of that class. To this end, C++ supports `friend` functions. A friend is not a member of a class but still has access to its private elements. Since a friend function is neither `private` nor `public`, it can be declared anywhere in the class definition. The following example demonstrates the syntax as well as how to make a function friend of more than one class.

```
class classB;

class classA {
  int a;
  friend bool is_equal(const classA &, const classB &);
};

class classB {
  int b;
  friend bool is_equal(const classA &, const classB &);
};

bool is_equal(const classA &A, const classB &B) { return (A.a == B.b); }
```

As a general rule of thumb, use member functions where possible, and friend functions only where necessary. In much the same way as you should make data and methods of your classes private where possible, and public where necessary.

2.10 Friend classes

In much the same way that we can make a (global) function a friend of a class, we can also declare another class to be a friend of a class. This means that the member functions of the friend class are allowed to access also the private member data of the original class. Here is a short example.

```
class classA {
  int a;
  friend class classB;
};

class classB {
  int b;
public:
  void convert(const classA &A) { b = A.a; }
};
```

2.11 Header files

In large programming projects it is often necessary to make use of header files. Usually, the header file will include all the information, that other parts of the project need to know in order to use certain functions and classes. Including a header file with the `#include` preprocessing directive, followed by the

name of the header file in inverted commas, is like copying and pasting the contents of the header file into the C++ source file. Below you can find a simple example.

Contents of "script_header.h"

```
#ifndef SCRIPT_HEADER_H
#define SCRIPT_HEADER_H

class A {
  int a;
public:
  A(int = 0);
  void print() const;
private:
  void check() const;
};
#endif  // SCRIPT_HEADER_H
```

Contents of "script_header.cc"

```
#include <iostream>
#include "script_header.h"
using namespace std;

A::A(int i) { a = i; }
void A::print() const { cout << a << endl; check(); }
void A::check() const { if (a == 5) cout << "5!!" << endl; }

int main() {
  A x(2), y(5), z;
  x.print(); y.print(); z.print();
  return 0;
}
```

Note that the macro definition inside the header file prevents the class to be defined twice, in case the header file is included in more than one source file. Note furthermore, that a header file usually only provides the declaration of member functions, whereas the actual implementation takes places in the C++ source file.

2.12 Strings revisited

Now that we are familiar with the concept of class member functions, we can have another look at the C++ string class. In §1.5 we already saw how to use the basic features of the C++ string class. In the following program we demonstrate the use of some other useful member functions of this class. For further details visit e.g. `www.cppreference.com/cppstring.html`.

```
#include <iostream>
#include <string>
using namespace std;

int main()  {
  string f("    Leading and trailing blanks      ");
  cout << "String f            : " << f << endl;
  cout << "String length       : " << f.length() << endl;
  cout << "String f            : " << f.append("ZZZ") << endl;
  cout << "String length       : " << f.length() << endl;
  cout << "substr(5,9)         : " << f.substr(5,9) << endl;
  cout << "assign(\"xyz\",0,3)   : " << f.assign("xyz",0,3) << endl;
  cout << "insert(1,\"abc\")     : " << f.insert(1,"abc ") << endl;

  string g("abc abc abd abc");
  cout << "String g            : " << g << endl;
  cout << "replace(12,1,\"xyz\") : " << g.replace(12,1,"xyz") << endl;
  cout << "replace(4,3,\"xyz\",2): " << g.replace(4,3,"xyz",2) << endl;
  cout << "replace(4,3,\"ijk\",1): " << g.replace(4,3,"ijk",1) << endl;
  cout << "find(\"abd\",1)       : " << g.find("abd",1) << endl;
  cout << "find(\"xyzb\")        : " << g.find("xyzb") << endl;
  return 0;
}
```

The above code produces the following output:

```
String f            :     Leading and trailing blanks
String length       : 37
String f            :     Leading and trailing blanks      ZZZ
String length       : 40
substr(5,9)         : eading an
assign("xyz",0,3)   : xyz
insert(1,"abc")     : xabc yz
String g            : abc abc abd abc
replace(12,1,"xyz") : abc abc abd xyzbc
replace(4,3,"xyz",2): abc xy abd xyzbc
replace(4,3,"ijk",1): abc iabd xyzbc
find("abd",1)       : 5
find("xyzb")        : 9
```

.

A further important member function is `string::c_str()` which returns a pointer to a character array with the same characters as the string that generates the call. This method is needed if you want to use some "older" library routines such as `sscanf`.

3 Overloading

As already indicated in §1.6 and §1.7.4, in C++ it is possible to define several functions with the same name, provided that the type of their arguments or the number of the arguments they take differ. After classes, this is perhaps the next most important feature of C++.

3.1 Pointers to overloaded functions

Sometimes it might be necessary to pass the address of an overloaded function to a pointer. E.g. when passing a function as a parameter to another function. The way to tell C++ which function's address we would like to access is to include the list of parameters that the function takes into the definition of the pointer. Below is a simple example.

```
#include <iostream>
using namespace std;

void print(int i) { cout << "i = " << i << endl; }

void print(double a, double b) { cout << "a = " << a << ", b = " << b << endl; }

int main() {
  void (*p)(int)            = print;
  void (*q)(double, double) = print;
  p(2);                              //  'i = 2'
  q(1.0,2.0);                        //  'a = 1, b = 2'
  return 0;
}
```

3.2 Overloading constructors

As we have seen in §2.4, a very common example of function overloading is to provide several different versions of a class's constructor. This is only natural, since a class needs a constructor for each way that an object of that class is declared in a program. In particular, when dynamically allocating memory for objects of a class, one has to define a default constructor that takes no arguments.
Here is an example for the `student` class.

```
student *p;
p = new student[1];               // calls student::student();
p->print();                       // 'unknown, unkown (YEAR 1)'

delete [] p;
```

3.3 Operator overloading

In C++, operators like `+`, `*`, `<`, `>>`, `[]` and `()` are just a very special type of functions. However, when overloading operators some special rules apply. In particular, it is not possible to change the number of arguments an operator takes. Furthermore, it is not possible to redefine an operator for the basic

types. I.e. when overloading an operator at least one of its arguments must be a newly defined class (or enumeration).
Hence an overloaded operator never loses its original meaning. Instead, it gains additional meaning relative to the class for which it is defined. Furthermore, even if an operator is overloaded, it maintains its precedence over other operators and its associativity.
When overloading an operator relative to a newly defined class, where an object of that class appears on the left hand side of the operator, there are two ways of defining the operator function. Either the operator is overloaded by a member function of the class, or the operator is defined as a global function. In the first case the operator function has to be passed one parameter less than the operator takes, since the call of the operator function will always be generated by the object on the left hand side of the operator. In the second case, the operator function takes as many parameters as the operator itself. Moreover, in general this global operator function will be declared as a `friend` of the class it is defined for, cf. §3.7.
Finally, operators may not have default parameters.
In the following example, the `[]` subscript operator of a class is overloaded.

```
#include <iostream>
using namespace std;

class A {
public:
  void operator[] (int i) {                              // overloading A[int]
    cout << "You want: [" << i << "]." << endl;
  }
};

int main() {
  A a;
  a[5];                  // 'You want: [5].'
  return 0;
}
```

Although it is permissible to have an operator function perform virtually *any action* (see example above), it is highly advisable to stick to operations similar to the operator's traditional use.

3.4 Overloading binary operators

When a member operator function overloads a binary operator, the function will have only one parameter. This parameter will receive the object that is on the right hand side of the operator. The object on the left hand side is the object that generated the call to the operator function.
Here is an example of overloading the assignment `operator=()`.

```
student &student::operator=(const student &s) {
  name = s.name; firstname = s.firstname; year = s.year;
  return *this;
}
```

Note the use of the `this` pointer and the fact that the `operator=()` function returns a reference to the object that something is assigned to. This is done in order to allow multiple assignments like `a = b = c;`, which are syntactically correct in C++.
Observe that for basic classes like `student` the assignment operator is already well defined, similarly to the copy constructor (cf. §2.4.1). This is due to the fact that the built-in assignment operator performs a bit-wise copy operation. Hence in practice one only needs to overload the assignment operator if the class has a pointer to memory that is dynamically allocated.
The following program demonstrates that it can be important to differentiate between the object that generates the call, and the object that is passed as a parameter. In this example the operators `-=` and `-` are overloaded.

```
#include <iostream>
using namespace std;

class Point {
  double x,y;
```

```
public:
  Point(double a = 0, double b = 0) { x = a; y = b; }
  Point &operator-= (const Point &p) { x -= p.x; y -= p.y; return *this; }
  Point operator- (const Point &p) const { Point r(x-p.x,y-p.y); return r; }
  void print() const { cout << x << "," << y << endl; }
};

int main() {
  Point p(1.0,2.0), q(0.0,0.5), r;
  r = p -= q;
  p.print();            // '1,1.5'

  r = p - q;            // '1,1'
  r.print();
  return 0;
}
```

Observe that the `operator-= ()` function could have been defined to return `void` instead. But then statements like `r = p -= q;` would no longer work.

3.4.1 Overloading relational operators

A special family of binary operators are the relational and logical operators. Naturally, they too can be overloaded for newly defined classes. Here is an example.

```
class Point {
  double x,y;
public:
  bool operator< (const Point &p) const {
    return (x < p.x || (x == p.x && y < p.y));
  }
  // etc
};
```

The C++ compiler treats the relational operators like any other binary operator. In particular, overloading the operator `<` does not give meaning to the operators `>`, `<=` and `>=`. That means that if you want to make use of them, you have to overload them individually.

3.5 Overloading unary operators

Overloading a unary operator is similar to overloading a binary operator. The only difference is that there is only one operand to deal with. Hence when overloading a unary operator using a member function, the function takes no parameters.
Some unary operators have both a *prefix* and a *postfix* version. In order to differentiate between these two when overloading the operator, the postfix version of the operator is defined as if it takes a (dummy) parameter of type `int`.
In the following example you can see how to overload unary operators, and in particular how to define both the prefix and postfix version of the `++` operator.

```
#include <iostream>
using namespace std;

class Point {
  double x,y;
public:
  Point(double a = 0, double b = 0) { x = a; y = b; }
  Point &operator++ () { x++; y++; return *this; }                  // prefix
  Point operator++ (int) { Point r(x,y); x++; y++; return r; }      // postfix
  Point operator- () const { Point r(-x,-y); return r; }
  operator double() const { return x; }                            // casting
  void print(ostream &os) const { os << "(" << x << "," << y  << ")"; }
};

ostream &operator<< (ostream &os, const Point &p) {
```

```
  p.print(os); return os;
}

int main() {
  Point p(0,0);
  cout << p++ << endl;      // '(0,0)'                    // postfix
  cout << p << endl;        // '(1,1)'

  cout << ++p << endl;      // '(2,2)'                    // prefix
  cout << p << endl;        // '(2,2)'

  cout << -p << endl;       // '(-2,-2)'                  // always prefix
  cout << p << endl;

  cout << (double) p << endl; // '2'
  return 0;
}
```

Note also the example for overloading the type casting operator, in the example above for the casting to type `double`.

3.6 Overloading the () operator

The `()` function call operator allows you to define functions as freely as you can outside classes. The function call operator can take any number of parameters and return any type you want. Here are two simple examples.

```
#include <iostream>
using namespace std;

class Point {
  double x,y;
public:
  Point(double a = 0, double b = 0) { x = a; y = b; }
  void operator() (double x) const {
    cout << "Function called with double x=" << x << "." << endl;
  }
  int operator() () const {
    cout << "Function called with ()." << endl;
    return 1;
  }
};

int main() {
  Point p;
  p(2.0);        // 'Function called with double x=2.'
  int i = p();   // 'Function called with ().'
  return 0;
}
```

3.7 Using friend operators

It is also possible to overload an operator relative to a class by using a (global) friend rather than a member function. In this case, a binary operator has to be passed both operands explicitly, while a unary operator has to be passed the single operand.
Friend operator functions are usually used when defining operations involving built-in types, where the built-in type is on the left hand side of the operator. Hence the call to the operator function cannot be generated by an object of your newly defined class. The most common example is to overload the inserter operator `<<`, where usually an object of type `ostream` is on the left hand side of the operator. See e.g. the following example.

```
#include <iostream>
using namespace std;
```

```
class Point {
friend ostream &operator<< (ostream &os, const Point &p);
friend Point operator- (const double &x, const Point &p);
  double x,y;
public:
  Point(double a = 0, double b = 0) { x = a; y = b; }
};

ostream &operator<< (ostream &os, const Point &p) {
  os << "(" << p.x << "," << p.y << ")"; return os;        // access to x and y
}

Point operator- (const double &x, const Point &p) {
  Point r(x-p.x, -p.y); return r;                          // access to x and y
}

int main() {
  Point z(0.5,1.5), r;
  r = 2.0 - z;
  cout << r << endl;    // '(1.5,-1.5)'
  return 0;
}
```

Note, however, that it is not really necessary to use friend operators. See the example in §4.5 for a way to keep the private member data of a class really private.

4 Inheritance

Inheritance is one of the most important features of C++. It allows classes to inherit properties from another class and hence enables you to build a hierarchy of classes, moving from the most general one to the most specific one.
When one class inherits another, it uses this general form:

```
class derived_class_name : access base_class_name {
  // etc
}
```

Here, `access` is one of three keywords: `public`, `private`, or `protected`. The access specifier determines how elements of the base class are inherited by the derived class. When the access specifier is `public`, all public members of the base become public members of the derived class. If the access specifier is `private`, however, all public members of the base become private members of the derived class. In either case, all private members of the base remain private to it and are inaccessible by the derived class.
Here is a simple example.

```
class base {
public:
  int a;
  void print() const { cout << a << endl; }
};

class sub : public base {
  int b;
public:
  void print() const { cout << a << "," << b << endl; }
  void printbase() const { base::print(); }
};
```

In this example we added the data `b` to the base definition of the class and redefined the `print()` member function. Observe the usage of the scope operator `::` to relate to a base class function with the same name as a (new) function in the derived class.
Inheritance from more than one parent class is possible, but is not covered in this course.

4.1 Protected members

To allow for private members in the base class that will be private and accessible in the derived class, the access specifier `protected` is provided. It is equivalent to the `private` specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Hence the full general form of a class declaration looks something like this:

```
class class_name {
  // private members
protected: // optional
  // protected members
public:
  // public members
}
```

How members of the base class can be accessed in the derived class is shown in the following table.

access level	base class members		
	public	protected	private
`: public`	public	protected	inaccessible
`: protected`	protected	protected	inaccessible
`: private`	private	private	inaccessible

Table 1: Accessibility of base class members in derived class.

4.2 Constructors, destructors and inheritance

When a base and derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. The destructor functions are executed in reverse order. I.e. the base class constructor is executed before the constructor in the derived class. The reverse is true for destructor functions.
Furthermore, it is possible for the derived class constructor to pass arguments to the base class constructor. This is done in the following fashion.

```
class base {
protected:
  int a;
public:
  base(int i) { a = i; }
  void print() const { cout << a << endl; }
};

class sub : public base {
  int b;
public:
  sub(int i, int j) : base(i) { b = j; }              // calls base constructor
  void print() const { cout << a << "," << b << endl; }
};
```

Note that data members that are inherited from the base class will always be initialized through the base class constructor. This guarantees that even data that is inaccessible in the derived class (cf. §4.1) is properly initialized when an object of the derived class is created.

4.3 Pointers to derived classes

In C++, a pointer declared as a pointer to a base class can also be used to point to any class derived from that base class. For example, with the above definitions of `base` and `sub`, the following statements are correct:

```
base *p; // base class pointer
base b(1);
sub s(1,2);
```

```
p = &b;  // p points to base object
p = &s;  // p points to derived object
```

Although you can use a base pointer to point to a derived object, you can access only those members of the derived object, that were inherited from the base class. This is because the pointer has knowledge only of the base class. It knows nothing about the members changed or added by the derived class.
Note that a pointer to a derived class cannot be used to access an object of the base class.
Finally, everything mentioned here about pointers to derived classes also holds true for passing arguments by reference. I.e. a function that expects a reference to a base class object can also be passed a reference to a derived class object. The following example illustrates that.

```
void show(const base &b); // reference to base class

int main() {
  sub s(1,2);
  show(s);       // pass reference to derived object
  // etc
}
```

4.4 Virtual functions

Virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. A virtual function is a class member function that is declared within a base class and redefined by a derived class. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by the derived class implements its operation as it relates specifically to the derived class and hence creates a specific method. In order to declare a function virtual, precede its declaration in the base class with the keyword `virtual`.
What makes a virtual function special and interesting is that it supports so called *run-time polymorphism*. That is, when a base class pointer (or a base class reference) points to a derived object that contains a virtual function and that virtual function is called by using that pointer, C++ determines which version of that function to call based upon the type of the object that is pointed to.
This idea is best illustrated with an example.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void print() const { cout << "base" << endl; }   // virtual !
};

class sub : public base {
public:
  void print() const { cout << "sub" << endl; }
};

class subsub : public sub {
public:
  void print() const { cout << "subsub" << endl; }
};

int main() {
  base b; sub s; subsub ss;
  b.print(); s.print(); ss.print();                  // "base", "sub", "subsub"

  base *p[3];                                       // pointers to base class
  p[0] = &ss; p[1] = &s; p[2] = &b;
  for (int i = 0; i < 3; i++) p[i]->print();        // "subsub", "sub", "base"
  return 0;
}
```

Observe that without declaring the `print()` member function in the base class as `virtual`, the last three outputs would all read `"base"`. Only by making the function `virtual` do we force the compiler to select

the appropriate member function for each object of the derived classes.
In case that a virtual function is not overridden in one of the derived classes, the version of the next highest class where the function is redefined is executed. E.g. on introducing the derived class

```
class subsubsub : public subsub {};
```

the following statements in `main()` would also lead to the output `"subsub"`.

```
subsubsub sss;
p[0] = &sss;
p[0]->print();                                          // "subsub"
```

Finally, the virtual nature of a base member function is also invoked, when the function is called inside another base member function. Here one can think of the `this` pointer being used, see §2.6. The `this` pointer inside a base member function is a pointer of type `base *`. When that pointer is used to call a virtual member function, C++ determines which version of that function to call, depending on the object it points to. The following example demonstrates this.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void print() const { cout << "base" << endl; }   // virtual !
  void show() const;                                       // non-virtual
};

class sub : public base {
public:
  void print() const { cout << "sub" << endl; }
};

void base::show() const { print(); }                       // or: this->print();

int main() {
  base b; sub s;
  s.show(); b.show();                                      // "sub", "base"
  return 0;
}
```

4.4.1 Pure virtual functions

Sometimes when a virtual function is declared in the base class, there is no meaningful operation for it to perform. I.e. its definition in the base class would merely be used to define the function's interface, while there would be no sensible method for it in the base class.
This situation is quite common because often a base class does not define a complete class by itself. Instead, it simply supplies a core set of member functions and variables to which the derived classes supply the remainder. When there is no meaningful task for the base class virtual function to perform, then any derived class must override this function. In C++ this is ensured by defining the concerned function as *pure virtual*.
A pure virtual function has no definition in the base class, only the function's declaration is included. To make a virtual function a pure virtual function, it has to be "assigned" the value 0, e.g.

```
class base {
public:
  virtual void print() const = 0;
  // etc
}
```

Some programmers do not like this somewhat arbitrary syntax and define a special preprocessor macro called `pure` as follows.

```
#define pure = 0
```

Then the previous definition of the pure virtual function `print()` would change to the following.

```
class base {
public:
  virtual void print() pure;
  // etc
}
```

Keep in mind that when a function is defined as pure virtual, it forces any derived class to provide an overriding definition.

4.4.2 Abstract classes

A class that contains at least one pure virtual function is called an *abstract class*. All other classes are called *concrete classes*. Since an abstract class contains at least one function for which no body exists, it is technically speaking an incomplete type. In particular, no objects of that class can be created. The only purpose of abstract classes is to be inherited by derived classes. Note, however, that it is still possible to use pointers (and references) to abstract base classes.
Here is an example.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void print() const = 0;   // pure virtual !
};

class sub1 : public base {
  int a;
public:
  void print() const { cout << "sub1: " << a << endl; }
};

class sub2 : public base {
  double b;
public:
  void print() const { cout << "sub2: " << b << endl; }
};


int main() {
  sub1 s1; sub2 s2;
  base *p[2];                       // pointers to base class
  p[0] = &s1; p[1] = &s2;
  for (int i = 0; i < 2; i++) p[i]->print();
  return 0;
}
```

4.4.3 Constructors and destructors

While constructor functions cannot be made virtual, destructor functions can. The restriction for constructors makes sense since constructors must always be called explicitly, by naming the exact class to construct. They can never be called by pointer or by reference. Moreover there is no need to declare a constructor as virtual, since on recalling §4.2, a constructor in a derived class always invokes the constructor of the next higher class.
Destructors, on the other hand, can and often should be made virtual. In particular, if a class contains virtual member functions, then the destructor should be declared virtual to ensure that an object of a derived class is always destructed properly, even when pointed to by a base class pointer.
Here is a simple example.

```
#include <iostream>
using namespace std;

class base {
```

```
public:
  base() { cout << "Constructing base." << endl; }
  virtual ~base() { cout << "Destructing base." << endl; }    // virtual !
  virtual void print() const { cout << "base" << endl; }
};

class sub : public base {
public:
  sub() { cout << "Constructing sub." << endl; }
  ~sub() { cout << "Destructing sub." << endl; }
  void print() const { cout << "sub" << endl; }
};

int main() {
  base *p = new sub;

  delete p;                                                  // calls ~sub !
  return 0;                        // "Destructing sub.", "Destructing base."
}
```

4.4.4 Pure virtual destructors

In order to avoid linkage errors at compile time, special care has to be taken when declaring a base class destructor as pure virtual. As the compiler will invoke *all* the destructors of a given class hierarchy when an object of a derived class runs out of scope, it will in particular also attempt to call the base class destructor. Hence in this particular case, we need to also provide an implementation of the pure virtual destructor member function. As the C++ syntax prohibits this definition inside the class definition, this needs to be done after the class definition. Below is an example.

```
#include <iostream>
using namespace std;

class base {
public:
  base() {}
  virtual ~base() = 0;                                  // pure virtual !
};

/*********************************************************/
/* without the next line, the code does not compile/link */
/*********************************************************/
base::~base() {}

class sub : public base {
public:
  ~sub() { cout << "Destructing sub." << endl; }
};

int main() {
  base *p = new sub;

  delete p;                                             // calls ~sub !
  return 0;                                             // "Destructing sub."
}
```

4.4.5 Inheriting virtual functions

When a virtual function is inherited, so is its virtual nature. That means that when a derived class inherits a virtual function from a base class and is then used to derive yet another class, the virtual function may be overridden by the final derived class. See also the example in §4.4.
Moreover, since the inherited function is virtual in the firstly derived class, a pointer to that class can also be used to implement run-time polymorphism. Here is an example.

```
#include <iostream>
using namespace std;

class base {
public:
  void print() const { cout << "base" << endl; }            // here not virtual!
  virtual void show() const { cout << "show base" << endl; }
};

class sub : public base {
public:
  virtual void print() const { cout << "sub" << endl; }     // now virtual
  void show() const { cout << "show sub" << endl; }         // virtual by default
};

class subsub : public sub {
public:
  void print() const { cout << "subsub" << endl; }
  void show() const { cout << "show subsub" << endl; }
};


int main() {
  base b; sub s; subsub ss; base *pb[3]; sub *ps[2];
  pb[0] = &b; pb[1] = &s; pb[2] = &ss;
  for (int i = 0; i < 3; i++) {                    // base, show base
    pb[i]->print(); pb[i]->show();                 // base, show sub
  }                                                // base, show subsub
  ps[0] = &s; ps[1] = &ss;
  for (int i = 0; i < 2; i++) {
    ps[i]->print(); ps[i]->show();                 // sub,    show sub
  }                                                // subsub, show subsub
  return 0;
}
```

4.5 Friend functions and inheritance

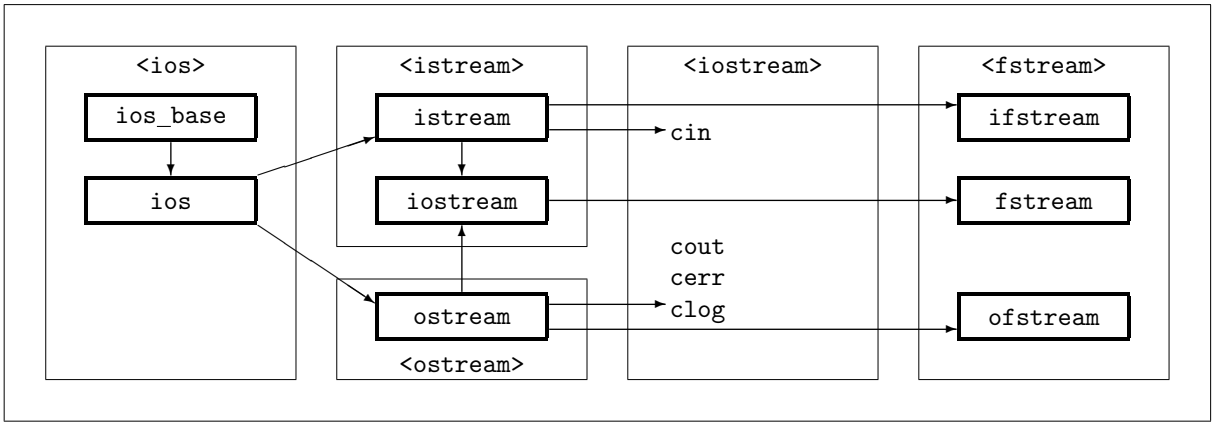
A friend function is not inherited. That is, when a base class includes a friend function, that friend function is not a friend of a derived class. A nice way around this problem is to put the main part of the function into a `virtual` (!) member function. This means in particular, that the function does not need to be declared as `friend` anymore. The following example demonstrates this using operator overloading, see §3 for details on that.

```
#include <iostream>
using namespace std;

class base {
protected:
  int a;
public:
  virtual void print(ostream &os) const { os << a; }
};

class sub : public base {
  int b;
public:
  void print(ostream &os) const { os << a << "," << b; }
};

ostream &operator<< (ostream &os, const base& b) { b.print(os); return os; }

int main() {
  base b; sub s;
```

```
  cout << "b = " << b << ", s = " << endl;
  return 0;
}
```

4.6 Example for inheritance

The `iostream` library is an object-oriented library that provides input and output functionality using streams. The following figure illustrates the dependence of the different classes provided by the library.



5 Templates

Templates are a very powerful feature of C++. They are a relatively new addition to C++, and they introduce the concept of generic programming. Generic programming is a data structure independent way of developing and delivering algorithms that need not be bound to a specific object.
In particular, C++ templates allow you to implement generic functions or classes independent of a certain parameter's type. Once a template is defined, it can be used in connection with objects from any type, provided that all the operations within the template are well defined for this type.
Templates are very useful when implementing generic constructs like vectors, stacks or lists, which can be used with an arbitrary type. The most famous example for this kind of application is the *Standard Template Library*, cf. §6.
C++ templates provide a way to reuse source code as opposed to inheritance and composition which provide a way to reuse object code. In C++ you can define two kinds of templates: class templates and function templates. The Standard Template Library, for example, uses function templates to define generic algorithms, and the containers are implemented as class templates.
More details on templates and many examples can be found at `www.josuttis.com/tmplbook`.

5.1 Function templates

Here is a very simple example that demonstrates the use of function templates. Observe that the syntax `template <class T>` is equivalent to the syntax `template <typename T>` used in this example.

```
#include <iostream>
using namespace std;

template <typename T>                // the following function uses the 'type' T
T absolute(const T &a) {
  if (a < 0) return -a;
  else return a;
}

int main() {
  int a = -5;
  double b = 1.2;
  float c = -0.43;
```

```
  long int d = -123456789;

  cout << "Absolute values are : " << absolute(a) << "," << absolute(b) << ","
       << absolute(c) << "," << absolute(d) << endl;
  return 0;
}
```

Note that for the above program the compiler creates four different versions of the function `absolute()` — one for each of the four types the function is invoked for.
Moreover, it is worth noting that one could have referred to the different overloaded versions directly, as in the following example.

```
cout << "Absolute values are : " << absolute<int>(a) << endl;
```

In the above example, this is not really necessary. However, this way of identifying the intended function has to be used, whenever at least one template `typename T` does not appear in the signature of the function. Here is a small example.

```
#include <iostream>
using namespace std;

template <typename T>
void foo(int i) {
  T local = 0;
  cout << local << " " << i << endl;
}

int main() {
  foo<double>(2);
  return 0;
}
```

5.2 Class templates

Similar to function templates, a class template definition looks like a regular class definition, except that it is prefixed by the keyword `template`. For example, the next program defines a class template `my_pair`.

```
#include <iostream>
using namespace std;

template <typename T> class my_pair {
  T a,b;
public:
  my_pair(T i, T j) : a(i), b(j) {}
  void print() const;
};

template <typename T>
void my_pair<T>::print() const {                // define print for my_pair<T>
  cout << a << "," << b << endl;
}

int main() {
  my_pair<int>    a(-1,2);
  my_pair<double> b(1.2,0.1);

  a.print(); b.print();
  return 0;
}
```

5.2.1 Inheritance

When inheriting from a class template, one has to specify which version of the base class one refers to. In particular, one could either inherit the full template, or just one instantiation. Here is an example for both.

```
#include <iostream>
using namespace std;

template <typename T> class base {                                  // base template class
protected:
  T a;
public:
  base(T i = 0) : a(i) {}
  void print() const { cout << a << endl; }
};

template <typename T> class sub : public base<T> {                  // refer to base<T> ,
protected:
  T b;
public:
  sub(T i, T j) : base<T>(i), b(j) {}                               // its constructor ,
  void print() const { cout << base<T>::a << " " << b << endl; } // and its data
};

class subsub : public sub<int> {                                    // refer to sub<int>
  int c;
public:
  subsub(int i, int j, int k = 0) : sub<int>(i,j), c(k) {}
  void print() const { cout << a << " " << b << " " << c << endl; }
};

int main() {
  subsub       a(-1,2);
  sub<double>  b(1.2,0.1);

  a.print(); b.print();
  return 0;
}
```

Here a short remark is due. Observe that in the member functions of the inherited class template `sub<T>` particular attention has to be paid on how to refer to member data that was inherited from `base<T>`. In this example, the correct syntax is

```
base<T>::a
```

which is slightly cumbersome. However, this is the only way we can refer to that member data. The reason behind that lies in the C++ standard for templates. Strictly speaking, when the member function in `sub<T>` is compiled, the base class itself has not yet been instantiated and as a consequence we have no knowledge of what member data it will provide. Referring to the desired member data directly with the help of the scope operator `::` circumvents this problem.

5.2.2 Template specialization

A template specialization allows a template to have specific implementations when the parameter type is of a determined type. For instance, in the following example the member function `mod()` returns the module between two integers if the class is `my_pair<int>` and 0 otherwise.

```
#include <iostream>
using namespace std;

template <typename T> class my_pair {
  T a,b;
public:
  my_pair(T i, T j) : a(i), b(j) {}
  T mod() { return 0; }
};

template <>
int my_pair<int>::mod() {          // overload definition for my_pair<int>
```

```
  return a % b;
}

int main() {
  my_pair<int>    a(-1,2);
  my_pair<double> b(1.2,0.1);
  cout << a.mod() << endl;          // '-1'
  cout << b.mod() << endl;          // '0'      by default
  return 0;
}
```

Of course, we can also use template specialization in order to identify the `typename` that our template was instantiated for. The following example demonstrates this.

```
#include <iostream>
using namespace std;

template <typename T> class my_pair {
  T a,b;
public:
  my_pair(T i, T j) : a(i), b(j) {}
  int type_id() const { return 0; }                    // used to identify T
  void print() const;
};

template <>
int my_pair<int>::type_id() const { return 1; }        // specialize for int

template <>
int my_pair<double>::type_id() const { return 2; }     // specialize for double

template <typename T>
void my_pair<T>::print() const {
  int t = type_id();
  switch (t) {
    case 1 : cout << "Two integers: "; break;
    case 2 : cout << "Two doubles: ";  break;
    default: cout << "Unknown type: "; break;
  }
  cout << a << " , " << b << endl;
}

int main() {
  my_pair<int>    a(-1,2);
  my_pair<double> b(1.2,0.1);
  my_pair<float>  c(-1.1,0.2);

  a.print();                                       // 'Two integers: -1 , 2'
  b.print();                                       // 'Two doubles: 1.2 , 0.1'
  c.print();                                       // 'Unknown type: -1.1 , 0.2'
  return 0;
}
```

5.2.3 Partial specialization

It is also possible to generate a specialization of the class for just one of several type parameters. Here is an example.

```
#include <iostream>
using namespace std;

template <typename T1, typename T2> class my_pair {      // base template class
  T1 a; T2 b;
public:
```

```
  my_pair(T1 i, T2 j) : a(i), b(j) {}
  void print() const;
};

template <typename T1, typename T2>
void my_pair<T1, T2>::print() const {
  cout << a << "," << b << endl;
}

template <typename T> class my_pair<T, double> {        // partial specialization
  T a; double b;
public:
  my_pair(T i, double j) : a(i), b(j) {}
  void print() const;
};

template <typename T>
void my_pair<T, double>::print() const {
  cout << "The second data member is of type double: " << a << "," << b << endl;
}

int main() {
  my_pair<int, int>       a(-1,2);
  my_pair<double, double> b(1.2,0.1);

  a.print(); b.print();
  return 0;
}
```

Note that the definition of any specialization is *independent* from the base template. I.e. in theory a specialization could implement very different methods, though this would be very bad programming practice indeed.

5.3 Compile time computations

Templates in C++ do not only offer compile time *polymorphism* as described above. One of the less exploited and rather theoretical features is compile time *computation*.
The following example computes factorials of constant integer expressions at compile time.

```
#include <iostream>
using namespace std;

template <int N>
class Factorial {
public:
    enum { value = N * Factorial<N-1>::value };
};

template <>
class Factorial<0> {
public:
    enum { value = 1 };
};

int main () {
  cout << "10! = " << Factorial<10>::value << endl;
  return 0;
}
```

The beauty of the above example is that no computation time is needed at run time.

6 STL

The Standard Template Library (STL) is a general purpose C++ library of algorithms and data structures. The STL is part of the standard ANSI-C++ library and is implemented by means of the C++ template mechanism, hence its name. While some aspects of the library are very complex, it can often be applied in a very straightforward way, facilitating reuse of the sophisticated data structures and algorithms it contains.
For a full documentation see `www.sgi.com/tech/stl`.

6.1 Vectors

The `vector` template is probably the most important and most widely used container class offered by the STL. What makes it so attractive is that one can use it in the same fashion as C style arrays without having to worry about memory management or possible memory leaks. Furthermore, by virtue of the `size()` member function, one knows at any point in time how many objects a vector contains. With C style arrays, of course, failure to monitor the size of an array properly can lead to incorrect programs and segmentation faults/fatal exception errors.
All of the STL container classes are treated by C++ almost like the built-in standard types. In particular, C++ takes care of the proper destruction of vectors and other containers as soon as one of these objects goes out of scope. Moreover, for the `vector` class basic operators like `=`, `==` and `<` are already defined and functional.
The following table gives an overview over the `vector` class's most important member functions. Here we assume that the size of the vector is of type `int`, that the objects contained in the vector are of type `T` and that the iterators are of type `T*`. Some of the type declarations will be omitted in the table.
Details can be found at `www.sgi.com/tech/stl/Vector.html`.

Member function	Description	Example
`begin()`	Returns iterator pointing to the beginning of vector.	`T *p = v.begin();`
`end()`	Returns iterator pointing to the end of vector.	`T *p = v.end();`
`size()`	Size of vector.	`if ( i < v.size() )`
`capacity()`	Number of elements for which memory has been allocated.	
`bool empty()`	True if vector's size is 0.	`if (!v.empty() )`
`operator[] (int i)`	Returns reference to the n^{th} element.	`v[2] = v[3];`
`at(int i)`	Same as `[]` but with boundary check. Not always available.	`v.at(2) = v.at(3);`
`vector()`	Default constructor: creates an empty vector.	`vector<T> v;`
`vector(int n)`	Creates a vector with `n` elements.	`vector<T> v(100);`
`vector(int n, const T &t)`	Creates a vector with n copies of `t`.	`vector<T> v(100, t);`
`vector(const vector&)`	Copy constructor.	`vector<T> u(v);`
`reserve(int n)`	Increase `capacity()` to `n`.	`v.reserve(10000);`
`push_back(const T&)`	Inserts a new element at the end.	`v.push_back(5);`
`pop_back()`	Removes the last element.	`v.pop_back();`
`swap(vector&)`	Swaps the contents of two vectors.	`u.swap(v);`
`insert(T *p, int n, const T& t)`	Inserts `n` copies of `t` before `p`.	`u.insert(u.end(), 9, t);`
`erase(T *p)`	Erases the element at position `p`.	`v.erase(v.begin() );`
`erase(T *f, T *l)`	Erases range `[f, l)`.	`v.erase(&v[3], &v[8]);`
`clear()`	Erases all of the elements.	`v.clear();`
`resize(int n, const T &t = T())`	Inserts/erases elements at the end so that size becomes `n`.	`v.resize(50);`
`operator= (const vector&)`	Assignment operator.	`u = v;`
`operator== (const vector&)`	Tests two vectors for equality.	`if (u == v)`
`operator< (const vector&)`	Lexicographical comparison.	`if (u < v)`

Table 2: Member functions of the STL `vector` container class.

Strictly speaking, the last two functions are no member functions. Instead they are implemented as global functions, taking *two* parameters of type `const vector&`.

6.2 Other sequential containers

The `vector` container is one of the *sequence containers* that the STL offers. The other sequence containers are `deque` and `list`. Note that the `vector` and `deque` classes support random access to elements, while the `list` class does not. That is why the STL `sort()` algorithm (cf. §6.6) does *not* work for the `list` class. Hence a sorting algorithm for lists is provided as a member function, see the example below.
The following table shows the insert and erase overheads of the containers `vector`, `list` and `deque`. You should keep these in mind when choosing a container for solving a specific task. E.g. a `list` is preferable when you do not need random access to elements and when you often have to erase elements in the middle, say.

container	insert/erase overhead		
	at the beginning	in the middle	at the end
`vector`	linear	linear	constant
`list`	constant	constant	constant
`deque`	constant	linear	constant

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

Here is a simple example for the `list` container class. Note that iterators will be covered in more detail in §6.4.

```
#include <iostream>
#include <list>
using namespace std;

int main() {
  list<int> L;
  L.push_back(0);
  L.push_front(0);
  L.insert(++L.begin(),2,3);

  L.push_back(5);
  L.push_back(6);

  L.sort();                          // the STL sort() algorithm does NOT work

  for(list<int>::iterator i = L.begin(); i != L.end(); ++i)  // linear traversal
    cout << *i << " ";
  cout << endl;
  return 0;
}
```

A second group of STL containers are the *container adaptors*: `stack`, `queue` and `priority_queue`. They were derived from the three original containers described above.

6.3 Associative containers

The last group are the so called *associative containers*. They include `set`, `multiset`, `map` and `multimap`. A `map`, for example, is a *sorted* associative container that holds pairs of keys and data, where each entry's key is unique. Hence it is ideally suited to implement dictionaries, for instance. Here is a short example for the `map` container class.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
  map<string, int> freq;                // map of words and their frequencies
  string word;

  while (cin >> word && word != "quit") {
    ++freq[word];
  }

  // write the frequency and the word
  for (map<string, int>::iterator i = freq.begin(); i != freq.end(); ++i) {
    cout << i->second << " " << i->first << endl;
  }
  return 0;
}
```

Details on all the STL containers can be found at the web site given before and also at e.g. `www.cppreference.com`.

6.4 Iterators

Iterators are a fundamental feature of the Standard Template Library. The concept of iterators was introduced in order to unify access to and traversal of different kinds of STL containers and, in particular, to allow most of the STL algorithms to work for any container type.
Let us look at a simple example.

```
#include <iostream>
#include <vector>
#include <list>
#include <map>
#include <string>
using namespace std;

int main() {
  vector<double> v(20,1.2);
  list<int> l(10,4);
  map<string, int> m;
  m["ab"] = 1; m["yz"] = 4; m["fg"] = 0;

  map<string, int>::iterator mitr;
  for(mitr = m.begin(); mitr != m.end(); ++mitr)
    cout << (*mitr).first << " " << (*mitr).second << endl;

  list<int>::iterator litr;
  for(litr = l.begin(); litr != l.end(); ++litr) cout << *litr << " ";
  cout << endl;

  vector<double>::iterator vitr;
  for(vitr = v.begin(); vitr != v.end(); ++vitr) cout << *vitr << " ";
  cout << endl;
  return 0;
}
```

Observe that the traversal of e.g. a `map`, a `list` and a `vector` can be done in exactly the same way, although the internal data structures of these containers are fundamentally different.
You can think of iterators as a special class that belongs to the STL containers and that is overloaded differently for each container. Through appropriately overloading operators such as `=`, `!=`, `++`, `*` (see above) and other operators, this iterator class provides access to the elements stored in a container and allows its traversal. Note that pointers are nothing other than special iterators for both C style arrays and STL vectors.
There are several advantages for the use of iterators. As you have seen, iterators can be uniformly used for all containers and also arrays. Furthermore, on creating your own container class with an appropriate iterator class, you can use the existing STL algorithms for your new class. Similarly, if you code a new (template based) algorithm for container classes that is written using iterators, it will apply to all existing containers for which there are iterators.
A simple example program, that outputs all the elements in a range defined by two iterators, is shown here:

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

template <typename T>
void show_range(T b, T e) {
  for (T i = b; i != e; i++) cout << (*i) << " ";
  cout << endl;
}
```

```
int main() {
  vector<double> v(11,1.2);
  list<int> l(5,4);
  int A[] = {1, 2, 3, 4, 5};
  show_range(v.begin(), v.end());
  show_range(l.begin(), l.end());
  show_range(A, A+5);
  return 0;
}
```

More details on iterators can be found at `www.sgi.com/tech/stl/Iterators.html`.

6.4.1 Constant iterators

If you want to use iterators within a function that takes an STL container as a constant parameter, i.e. as a parameter that is passed by a constant reference or by a constant pointer, then you will not be able to use standard iterators. That is because iterators can be used to read and write data inside a container. Fortunately, for this purpose constant iterators exist. See the example below.

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

int main() {
  vector<double> v(20,1.2);
  list<int> l(10,4);

  list<int>::const_iterator litr;
  for(litr = l.begin(); litr != l.end(); ++litr) cout << *litr << " ";
  cout << endl;

  vector<double>::const_iterator vitr;
  for(vitr = v.begin(); vitr != v.end(); ++vitr) cout << *vitr << " ";
  cout << endl;
  return 0;
}
```

Moreover, in any of the previous examples you may replace `iterator` with `const_iterator`, and the programs will work as before.

6.5 Functors

Many of the STL algorithms (cf. §6.6) can be given so called *function objects*, or *functors*, as one of their parameters. A functor is simply any object that can be called as if it is a function. Hence an ordinary function is a functor, and so is a function pointer. Moreover, any object of a class that defines an `operator()` is a functor.
The STL offers many predefined generic functors, such as `plus`, `minus`, `multiplies`, `divides` and `modulus`. Furthermore, many `bool` functors – so called *predicates* – are defined within the STL as well. E.g. `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` and `less_equal`.
The following program shows how some of these templates can be used.

```
#include <iostream>
#include <cmath>
#include <functional>
using namespace std;

struct abs_greater : public binary_function<double, double, bool> {
  bool operator()(double a, double b) { return (abs(a) > abs(b)); }
};

int main() {
  double a = -5.1, b = 3.4;
```

```
  abs_greater a_g;
  plus<double> pl;

  if (a_g(a,b)) cout << "|a| > |b|" << endl;
  cout << "a + b = " << pl(a,b) << endl;
  return 0;
}
```

Of course, as stand alone classes these functors do not make much sense. But in connection with the STL algorithms, they can be very powerful.

6.6 Algorithms

The STL also includes a large collection of algorithms that manipulate the data stored in containers. The following example demonstrates the use of some of the STL algorithms. Although the program uses a `vector<int>` container, most of the algorithms also work for many other STL containers. To make this possible, all the algorithms are defined as global functions. Secondly, most of the algorithms are given a *range* to operate on. E.g. a full `vector<int> v` is given by the range `(v.begin(), v.end())`.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <iterator>
using namespace std;

int dice() { return rand() % 6 + 1; }

int main() {
  vector<int> v(1000);
  generate(v.begin(), v.end(), dice);                                    // generate
  cout << "Number of sixes: " << count(v.begin(), v.end(), 6) << endl; // count

  vector<int>::iterator firstfive = find(v.begin(), v.end(), 5);        // find
  cout << "Number after first five = " << *(firstfive+1) << endl;

  reverse(v.begin()+10, v.begin()+100);                                  // reverse
  random_shuffle(v.begin(), v.end());                                    // random_shuffle

  sort(v.begin(), v.end());                                              // sort with '<'
  sort(v.begin(), v.end(), greater<int>() );                             // sort with greater

  double a = max(1.2, 4.3);                                              // max;
  cout << "max(1.2,4.3) = " << a << endl;

  vector<int>::iterator min_el = min_element(v.begin(), v.end());       // min_element
  cout << "The smallest element is " << *min_el << endl;

  vector<int> u(10);
  for (unsigned int i = 0; i < u.size(); ++i) u[i] = i - 4;

  while (binary_search(u.begin(), u.end(), 0)) {                         // binary_search
    cout << "There exists a 0 element." << endl;
    vector<int>::iterator zero = find(u.begin(), u.end(), 0);
    u.erase(zero);
  }

  vector<int>::iterator new_end = remove(u.begin(), u.end(), -2);       // remove
  u.erase(new_end, u.end());

  copy(u.begin(), u.end(), ostream_iterator<int>(cout, " "));           // copy
  cout << endl;                                                          // '-4 -3 -1 1 2 3 4 5'
```

```
  cout << "The sum of all elements is "
       << accumulate(u.begin(), u.end(), 0) << endl;                    // accumulate
  cout << "The product of all elements is "
       << accumulate(u.begin(), u.end(), 1, multiplies<int>()) << endl;
  return 0;
}
```

The remarkable thing about the STL algorithms is that they work with almost any kind of container, including C style arrays. Here is an example, that first sorts and then outputs a given array.

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  sort(A, A + N);                                          // sort
  copy(A, A + N, ostream_iterator<int>(cout, " "));        // copy : "1 2 4 5 7 8"
  cout << endl;

  return 0;
}
```

7 Miscellaneous

7.1 Namespaces

Namespaces allow you to group a set of global classes, objects and/or functions under a name. This can be useful when writing large programs, where one wants to use the same name for two different objects in different contexts, say. If one chooses not to use the `using namespace` keyword, one can still refer to the defined objects with the help of the scope operator `::`. Here is an example.

```
#include <iostream>
using namespace std;

namespace first  { int var = 5; }
namespace second { double var = 3.1416; }

int main () {
  {
    using namespace first;
    cout << var << endl;
  }
  {
    using namespace second;
    cout << var << endl;
  }
  cout << first::var + second::var << endl;
  return 0;
}
```

When using the ANSI-C++ compliant include files one has to bear in mind that all the included functions, classes and objects will be declared under the `std` namespace.

7.2 Advanced I/O

Here is an example that demonstrates some of the advanced features of output operations under C++.

```
#include <iostream>
using namespace std;
```

```
int main() {
  cout << 123.23 << endl;                                  // '123.23'
  cout.setf(ios::scientific | ios::showpos | ios::uppercase);
  cout << 123.23 << endl;                                  // '+1.232300E+02'
  cout.unsetf(ios::showpos | ios::uppercase);
  cout << 123.23 << endl;                                  // '1.232300e+02'
  cout.precision(1);
  cout << 123.23 << endl;                                  // '1.2e+02'
  cout.precision(5); cout.width(13);
  cout << 123.23 << endl;                                  // '  1.23230e+02'
  cout.precision(5); cout.width(13); cout.fill('0');
  cout << 123.23 << endl;                                  // '001.23230e+02'
  cout.precision(3); cout.width(13);
  cout << 123.23 << endl;                                  // '00001.232e+02'
  cout << 123.23 << endl;                                  // '1.232e+02'
  return 0;
}
```

Of interested could also be the flags `ios::right` and `ios::fixed`.
For more details visit `www.cppreference.com/cppio.html`.
Another convenient way to simplify I/O operations is to define your own manipulators. Here is a short example.

```
#include <iostream>
using namespace std;

ostream &myformat(ostream &os) {
  os.width(10); os.precision(4); os.fill('*');
  return os;
}

ostream &tri(ostream &os) {
  os << "The result is: ";
  return os;
}

int main() {
  cout << tri << myformat << 123.23 << endl;     // 'The result is: *****123.2'
  return 0;
}
```

7.3 Static class members

If a class member is declared as `static`, only one copy of it is ever created, no matter how many objects of that class are created. In particular, the programmer has to define the variable also outside the class, so that memory is allocated for it. The following example demonstrates this.

```
#include <iostream>
using namespace std;

class myclass {
public:
  static int i;
  myclass(int ii) { i = ii;}
};

int myclass::i = 10;

int main() {
  myclass a(5);
  myclass b(8);
  cout << a.i << endl;            // '8'
  return 0;
}
```

7.4 Conversion functions

In C++ it is possible to define a conversion function that is used whenever an object of this class is found in a place where a different type is expected. Usually this will be one of the built-in C++ types. Here is an example.

```
#include <iostream>
#include <cmath>
using namespace std;

class cmplx {
  double r, i;
public:
  cmplx(double a, double b) : r(a), i(b) {}
  operator double() { return sqrt(r*r + i*i); }
};

int main() {
  cmplx c(2.0,1.5);
  double a = c;                                  // conversion to double
  cout << a << endl;                             // '2.5'
  return 0;
}
```

Obviously, one should use these conversion functions carefully.

7.5 Exception handling

Exception handling can be a useful tool when developing and debugging a program. The three commands used for it are `try`, `throw` and `catch`.

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  try {
    double *m;
    m = new double[10];
    if (m == NULL) throw string("Allocation failure");
    for (int i = 0; i < 100; i++) {
      if (i > 9) throw i;
      m[i] = i * i;
    }
    // etc
    delete [] m;
  }
  catch (int n) {
    cout << "Exception: index " << n << " is out of range." << endl;
    return 1;
  }
  catch (string &s) {
    cout << "Exception: " << s << endl;
    return 2;
  }
  return 0;
}
```

If you want to catch *any* exception within a `try` block, use `catch(...)` (i.e. three single dots within the brackets). Finally note that exceptions can also be thrown by functions called within a try block. Below is a simple example.

```
#include <iostream>
#include <string>
using namespace std;
```

```
double ratio(double a, double b) {
  if (b == 0.0) throw string("Division by zero");
  return a / b;
}

int main () {
  try {
    while (1) {
      double x, y;
      cout << "x, y = "; cin >> x >> y;
      cout << " x / y = " << ratio(x,y) << endl;
    }
  }
  catch (string &s) {
    cout << "Exception: " << s << endl;
    return -1;
  }
  return 0;
}
```

7.6 Temporary objects

In C++ temporary objects are not only created when functions return objects and when functions are passed objects by value. C++ also allows you to define your own temporary objects of your newly defined classes. These temporary objects are only valid for the line of code their appear in.
Here is a simple example.

```
#include <iostream>
#include <vector>
using namespace std;

class A {
  int a,b;
public:
  A(int i = 0, int j = 0) : a(i), b(j) {}
  void print() const { cout << a << "," << b << endl; }
};

void show(const A &x, const A &y) {
  cout << "x = "; x.print();
  cout << "y = "; y.print();
}

int main () {
  show(A(), A(1,2));
  vector<A> v(10, A(4,8));
  for (unsigned int i = 0; i < v.size(); i++) v[i].print();
  return 0;
}
```

Note that in the above a temporary object is used to initialize a `vector<A>`.

7.7 Passing functions as arguments

On recalling §3.1, it is straightforward to pass functions as arguments to other functions. Here is an example.

```
#include <iostream>
using namespace std;

double abs(double x) { return x > 0 ? x : -x; }

void print(double x) { cout << "Number is: " << x << endl; }
```

```
int combine(void (*f)(double), double (*g)(double), double x) {
  if (f != NULL && g != NULL) {
    f( g(x) );
    return 0;
  }
  else return 1;
}

int main() {
  return combine(print, abs, -2.5);
}
```

See also §1.7.1 for another example.