Chapter 3++

Basic Objects Usage and Definition

In this chapter we learn how to define objects and how to use them in Object Oriented Programming in C++. We do that working out the same example found in Chapter 3 of the C textbook: temperature conversion. Despite the syntax is the same, C++ is very different, in terms of concepts, with respect to C. It is, in fact, a completely different language. You should not consider it as an *evolution* of the C. It compares to C as transliterated chinese is similar to english.

C++ concepts are much more abstract than those introduced in the C language and it is much easier to illustrate them, profiting of the fact that you are assumed to know the C language.

3++.1 A basic object.

A C++ program, as an Object Oriented Programming language application, does not involve data manipulated by instructions, but *objects* who *interact* between themselves. Objects store their *status* and are responsible for changing it as well as for exposing it to other objects in the proper format. What does it mean for our example? In our example application, we want to be able to manipulate temperatures in different scales. Temperatures can be assigned by the user via the keyboard and shown to the user on the screen. Temperatures can be shown and assigned in any scale. In this context, temperatures are objects who own their state. The state of a temperature object is its value, regardless of the scale. A gas in a volume V and pressure p has its own temperature T that is fixed, despite the fact that it can be expressed in many different scales. The status of the gas at a given temperature T is independent on the choices about the units. In our program the user interact with the temperature object asking it to change status from a default temperature value to another temperature value, in any supported scale. The user can also ask the object to tell him the temperature in any supported scale. Whatever the scale, the temperature is always the same and represents the status of the object.

The user cannot access the status of the object directly. The access to the status is always mediated by the object itself. This property is known as *encapsulation*: the object encapsulates its own status and allows external objects to access it via some interface that regulates how to obtain information about it. In other words encapsulation means hiding to the user the way in which the status of an object is represented in memory. The status of an object, in fact, can be usually represented in many different ways. However, the choice about the representation, made at the time of the object definition, is of no interest for the user. The user is only interested in knowing the state of the object.

Let's see in practice how do we do that in C++, looking the the code in Listing 3++.1. In this listing we included two files: `iostream` and `Temperature.h`. The first is needed for I/O: it contains the definition of objects representing the standard input and standard output devices. `Temperature.h` contains the declaration of the *interface* of the `Temperature` object. The interface of an object is a sort of list of the properties of all the objects belonging to the same *class*. It specify the capabilities of the objects in terms of what are the allowed operations on them (*methods*) and how its status is represented internally (*attributes* or *members*).

```
#include <iostream>
#include <Temperature.h>
main() {
  Temperature t;
  double tf;
  std::cout << "Insert temperature in Fahrenheit: ";
  std::cin >> tf;
  t.setF(tf);
  std::cout << "Temperature in degree Celsius is: "
                      << t.getC() << std::endl;
}
```

Listing 3++.1 Temperature conversions.

In this program we *instantiates* an object of class `Temperature` and we call it `t`. The instantiation of an object is analogous to the declaration of a variable in procedural programming (in fact it is much more, but we discuss this aspect later in the book). We then use the object `std::cout`, defined in the `iostream` library to write on the screen the content of a string. We pass the string to the object using the *insertion operator* `<<`. The value of the temperature to be assigned to the object `t` is read from the keyboard, represented by the object `std::cin`, defined in `iostream` too, to which we apply the *extraction operator* `>>`. The value extracted from the standard input is inserted in the variable `tf`: a floating point number representing the value of the temperature in the given scale.

In order to instruct the object `t` to assume the temperature represented by `tf` we ask it to execute its method `setF`. A method works much like a function (see Chapter 7). It is an operation executed by the object specified before the *resolution operator* `.`.

The syntax `t.setF(tf)` asks the object `t` to execute its method `setF` that is meant, in this case, to assign the temperature of the object specified in degrees Fahrenheit within the parenthesis. The consequence of this operation is that the object `t` assumes the temperature specified by the user as degrees Fahrenheit. However, the status of the object is independent on the scale chosen to assign its temperature. In fact, in the following instruction, we ask the object to show its temperature in another scale, without the need to change it. The method `getC()`, in fact, returns the temperature of the object in degrees Celsius. Note also that the insertion operator can be catenated.

Of course there is nothing magic in it. The transformation between scales is done by the object and is defined in the implementation of the class `Temperature`. There is still much work to do for this example to work, probably much more work than those required for a procedural program. The advantage here is that now the behavior of the object is fixed by its interface and that its status is encapsulated within it. Concepts used in programming are then much better defined and much more similar to what we want to represent. Moreover encapsulation guarantees that programmers can only act on the status of an object throughout well defined operations and do not risk to change it in an uncontrolled way because they assigned the wrong value to a variable. There are many other advantages in OO programming, discussed later. It is important to know that simple applications are much more suitable to be realized in procedural programming languages, but as soon as the complexity of the program increases, OO programming techniques are much better, both in terms of simplicity as well as in terms of data management and control. In this context the complexity is a measure of how many relationships exist between the data to be manipulated.

3++.2 Declaring classes.

Let's see how to define the behavior of a class of objects and how OO programming allows the separation between the status of an object and its internal representation in terms of variables.

Classes are usually *declared* in files called after the name of class: in our example `Temperature.h`. This isn't strictly necessary (everything can be written in the same file), but is a very good practice. The content of the file `Temperature.h` for our example is listed in Listing 3++.2.

```
#ifndef _TEMPERATURE_H_
#define _TEMPERATURE_H_

class Temperature {
public:
  Temperature();
  void setF(double tf);
  double getC();
```

```
private:
  double _t;
};

#endif
```

Listing 3++.2 A prototype for the `Temperature` class.

First of all note the usage of the preprocessor directives `#ifndef`, `#define` and `#endif`. These directives allows programmers to freely include files in their applications, without the need to consider their ordering or the possibility to include one of them more than once. In fact, if `Temperature.h` is included in a program once, the preprocessor finds that the symbol `_TEMPERATURE_H_` is not defined and proceeds, defining that symbol and passing the rest to the processor. Once the program includes the file a second time (even via other included files), the preprocessor will find that the symbol `_TEMPERATURE_H_` is already defined, so it skips the rest of the file. It is then a good practice to start the implementation of a class writing the three preprocessor directives.

The C++ code follows. The class is declared using the keyword `class` followed by its name and a pair of braces, containing the details. The class definition ends with a mandatory semi–colon. Again, it is a good practice to start defining the class typing the two braces and the semi–colon. Forgetting it is a very common mistake, sometimes difficult to find from the compiler error report.

The class body is divided into two sections: `public` and `private`. In the public section we list all the operations (methods) that the objects of this class are able to perform, namely `setF()` and `getC()`. The first operation will not result in any object and accept a floating point number as input. The second returns a floating point number and does not require any input. Methods' declaration is much like function declaration. Besides those methods there is one method called the *constructor* that has no type and has the same name of the class. This method must exist and must have exactly that signature: i.e. the same name of the class and no formal parameters. The methods listed in the public section are those that can be called by the programmer on objects of type `Temperature`. We have not yet defined their behavior. We have just listed them and defined their signature.

The private section is not directly accessible to objects not belonging to the same class. From the main program we cannot access methods and data listed here. Usually members are listed here. Members are objects and variables that are used to represent the status of the object. In this case we choose to represent the temperature as a floating point number in a given scale (let's say the Celsius one). Since this member is inaccessible from outside, this choice is completely arbitrary, however it does not harm the working of the class. As a consequence, we see in a moment that even if we change the way in which we choose to describe the status, there is no consequence for the main program, nor for any other object using objects of the class `Temperature`. Simply stated, the *variable* `_t` represents the status of the object as the temperature given in degrees Celsius. Note that we called this member `_t` and not `t`. The leading underscore is used to distinguish member variables from other local variables used in the class definition. Again, this is not mandatory for C++, but it is a good practice. Some programmer does not like variable names starting with and underscore and another convention uses `m_` as the leading characters for members.

3++.3 Defining classes.

The definition of the behavior of methods is usually done inside a file called after the name of the class with extension `.cc`: `Temperature.cc`. First of all we need to include its declaration, then we list all the methods, specifying the class to which they belong and the set of instructions to be executed when the method is called. Let’s analyze Listing 3++.3.

```
#include <Temperature.h>

Temperature::Temperature() {
  _t = 0.;
}

void Temperature::setF(double tf) {
  _t = (tf - 32.) * 5./9.;
}

double Temperature::getC() {
  return _t;
}
```

Listing 3++.3 Definition of methods of the `Temperature` class.

The reason for which we need to specify the class name for each method (using the double colon `::` as a separator), is that C++ allows to define methods in any point of the program and in any file, so the compiler need a way to know which class is compiling.

We first implement the constructor method (the order is unimportant, in fact. This is another good practice). This method is called automatically as soon as an object is instantiated in a program. In our main program, this method is called as soon as the instruction `Temperature t;` is executed. Note that, contrary to C, this is not a mere declaration, but is in fact an instruction, since it implies the execution of the constructor. For this reason not only we are not forced to declare objects prior to any executable instruction, as in C, but we are encouraged to *declare* objects just before needed.

The aim of the constructor method is to execute all the necessary instructions to set up the object and to assign it a default status. In this case we chose to assign a temperature equal to 0°C by default. This is done assigning the value 0 to the variable representing the status
t _t.

The aim of the method `setF()` is to assign the temperature to the object, expressing it in degrees Fahrenheit. Since we chose to represent the status of the object as a number representing the temperature in degree Celsius, we need to transform the input value in this scale before assigning it to the member `_t`.

The method `getC()` just returns the member variable representing the temperature, since it is already given in the requested scale.

Let's discuss what happen if we change our mind and decide to represent the temperature of the object as an integer representing the temperature in mK. In this case `_t` is an `integer`. The `setF()` method changes like

```
void Temperature::setF(double tf) {
  _t = (int)rint(1000.*((tf - 32.) * 5./9. + 273.15));
}
```

We always accept a floating point number representing the temperature in degrees Fahrenheit as input, but we transform it in Kelvin first, adding 273.15, then in mK multiplying by 10^3. We used the `rint` function, defined in `math.h`, to be included, to round the result to the nearest integer and we cast the result to an integer. Accordingly, the `getC()` method changes as

```
double Temperature::getC() {
  return (double)_t/1000. - 273.15;
}
```

From the point of view of a *user* of the Temperature class, nothing has changed. The main program remains unchanged. This is the result of the encapsulation: whatever the choices about the internal representation of the status of an object, the latter is independent on that choice. The temperature of an object is always the same, despite it is represented in a way or in another. What is important is that the status appears always as the same, not that it is represented always in the same way.

3++.4 Extending class behavior.

A well designed class must hide all the details about implementation choices, provide all the methods that, in principle, can be called by other objects and forbid usages that are not within scoper of the class. A very good news for C++ programmers is that this language provides many levels of *polymorphism*. Polymorphism is the ability of an object to behave differently according to the context. A very basic kind of polymorphism is *overloading*: a feature that allows the definition of methods with exactly the same name that behaves differently according to the way in which they are called. Of course there must be a way for the compiler to choose the right sequence of instructions when transforming a method call into machine language. The way in which this is provided is by analyzing the method prototype: methods are distinguished by their signature in terms of formal parameters.

Another nice feature of C++ programming is the ability to redefine operators. The behavior of operators like `+`, `-=`, `>`, etc., is very useful expecially for scientific programming. Unfortunately no new operators can be defined, so, for example, despite the strive for operators for power raising, scientists must continue using functions to perform this operation.

We now see method and operator overloading in action, looking at Listing 3++.4.

```
#ifndef _TEMPERATURE_H
#define _TEMPERATURE_H

class Temperature {
/*
  This class represents temperatures. The default scale is Kelvin.
*/
public:
  Temperature() { _t = 0.; }
  Temperature(double tk);  // constructor overloading
  ~Temperature();

  void setF(double tf); // set temperature in Fahrenheit
  void setC(double tc); // set temperature in Celsius
  void setR(double tr); // set temperature in Reamur
  void setK(double tk); // set temperature in Kelvin

  double getF();
  double getC();
  double getR();
  double getK();

  // operator overloading
  Temperature& operator=(const Temperature &tr);
  Temperature& operator=(double tk);
  Temperature operator+(const Temperature &tr);
  Temperature operator+(double tc);
  Temperature operator-(const Temperature &tr);
  Temperature operator-(double tc);
  // note that the multiplication of two temperatures is not a temperature
  Temperature operator*(double C);
  Temperature operator/(double C);
  // the ratio between two temperatures is a number
  double operator/(const Temperature &tr);
  // note the & sign
  Temperature& operator+=(const Temperature &tr);
  Temperature& operator-=(const Temperature &tr);
  Temperature& operator*=(double C);
```

```
  // note that in this case no division by temperature is supported
  Temperature& operator/=(double C);
 private:
  double _t;
};

#endif
```

Listing 3++.4 A more complete `Temperature` class.

As you can see in this class we defined two constructors. They differ because of the signature. The (mandatory) default constructor with no parameters and an overloaded constructor with a floating point number as a parameter. You may notice that the definition of the default constructor has been written close to it, rather than in the `.cc` file. This style is common for very short methods and makes the method *inline*, i.e. the compiler does not generate a function to which the execution of the program jumps once the method is called, but repeat the same set of instructions at each point of the program in which the method is invoked. In this case we must not put the same code in the definition file. The code of the overloaded constructor, found in the `Temperature.cc` file reads

```
Temperature::Temperature(double tk) {
  _t = tk;
}
```

With this constructor the default temperature of the object becomes `tk` degrees Kelvin, while constructing the object as

```
  Temperature t(100.);
```

It is worth noting that there is no way here to establish a different temperature scale, since in another scale the constructor will look like almost the same. The opportunity of providing such a constructor, then, can be discussed. However, once decided, there should be some hint for the programmer about how to use it. In our case we provide two different kind of hints: comments within `/* ... */`, instructing the programmer that the chosen default scale is the absolute one, and the fact that the variable is called `tk`. Comments in C++ can be written after a double slash, too: `//`. This syntax is used mostly to write short comments as in few cases in Listing 3++.4.

You may also notice the presence of a *destructor*: a method called automatically when an object is removed from the memory. The destructor method must have the same name of the class prefixed with a `~` sign. In our case the object is destructed at the end of the program. Usually the destructor may be left blank or even undefined, unless some special operation must be done at the destruction time.

After constructor and destructors, we declare methods used to assign the temperature of the object in different scales, as well as methods to get the temperature of the object in the same scales. The implementation of those methods is left as an exercise.

What follows is the list of operators defined for this object. Operators are defined as methods whose name is `operator` followed by the operator symbol followed by the right operand, if applicable. Operators can be overloaded, too. For example, consider the first two operators:

```
Temperature& operator=(const Temperature &tr);
Temperature& operator=(double tk);
```

These methods define two versions of the same operator `=`. The first define the assignment operator when the rightmost operand is a temperature object; the second define the assignment operator when the rightmost operand is a number. The leftmost operator is, by definition, a temperature object. Consider the following piece of code:

```
Temperature t1(100.);
Temperature t2 = t1;
Temperature t3 = 273.15;
```

The first line construct a temperature object and assign it a temperature of 100°K. The second line calls the assignment operator. Since the right hand operand is an object of class `Temperature`, the called method is the first version. When writing an operator you can imagine that, from the point of view of the compiler, the instruction containing the assignment operator is transformed into

```
t2.operator=(t1);
```

The signature of the method `const Temperature &tr` instructs the compiler that the object on the right of the operator must not change during the execution (`const`). The ampersand sign before the name of the formal parameter instruct the compiler to pass the address of the parameter in the memory rather than a copy of the object. To fully understand this syntax you must know about pointers, discussed later in the book. For the time being, just consider this as a rule, useful when the object on the right of an operator is rather complex. For simple arguments, such as in the second version of the operator, there is no need to use such a syntax. As a rule of thumb you may consider to use the `const` *className*`&` *objName* style when the parameter is an object and the standard style when the parameter is a variable. This latter version of the operator is called to execute the instruction `Temperature t3 = -273.15;` that, in fact, consists in the construction of an object called `t3` and the assignment of its temperature to the value 273.15°K. Note that, analogously to the constructor behavior, we assume that the default scale is the absolute scale. Again one can decide not to provide such an operator, so a programmer cannot use such a syntax and must assign the temperature using a `setX()` method or the assignment operator in the first form only.

The definition of the two methods reads:

```
Temperature& Temperature::operator=(double tc) {
  _t = tc;
  return *this;
}
```

```
Temperature& Temperature::operator=(const Temperature &tr) {
  _t = tr._t;
  return *this;
}
```

Both methods returns an object of class `Temperature&`. The ampersand at the end of the class name means that what is returned is in fact the address of the object and this is mandatory to ensure that the operator can be catenated. Omitting the ampersand will result in returning, in fact, a copy of the object. Again, such a syntax can be understood after learning pointers. In practice, the ampersand at the end guarantees that instructions like

```
t2 += t *= 2;
```

where `t2` and `t` are objects of class `Temperature`, are possible. This instruction means: the status of the object `t` must be those corresponding to doubling its temperature. In turns, the object `t2` must assume the same status of the `t` object. As a rule of thumb, one can say that the ampersand must be added to the object returned by the operator, if the operator is intended to modify the status of the operand on the left, as in this case, and can be catenated to other operators. In fact, even `setX` methods could be defined like this:

```
Temperature& setC(double &tc);
```

With such a definition the programmer is allowed to execute instructions such as

```
t.setC(30)*=2;
```

that means: assign to `t` the status corresponding to a temperature of 30°C, then multiply the temperature of this object by 2. Remember, in fact, that methods are executed left to right.

Looking at the implementation of the methods one can see that the status of the calling object is assigned assigning the value passed as an operand `tr` to its member `_t`. The attribute `this` is automatically created by the compiler for any class and is equal to the address of the object in the memory. `*this` returns then the object itself (see the chapter on pointers). This is why the methods can be catenated: if a method returns the calling object, the dot operator `.` acts on it again once executing the next method in the sequence.

Let's discuss briefly other operators. The `+` operator exists in two versions, too. Note that it returns an object of class `Temperature` without the ampersand since an instruction like `t3 = t1 + t2;` is not meant to modify the status of `t1`. With these definitions we establish that it is possible to sum two temperature objects and that the result is a temperature object, as well as it is perfectly legal to sum a temperature object (the leftmost operand) to a number (the rightmost one). The result of this operation is still a temperature object. Note that we cannot define operators that allows summing a number, as the leftmost operator, to a temperature, i.e. the `+` operator does not commute in C++.

Since the `+` operator does not modify the leftmost object, it must return a different object, so its implementation reads like:

```
Temperature Temperature::operator+(const Temperature &tr) {
  Temperature ret;
  ret._t = _t + tr._t;
  return ret;
}
```

Besides operators we provide an operator that allows the multiplication between a temperature and a number. We do not provide an operator for multiplying a temperature by another temperature since the result is not a temperature, neither belongs to any existing class or type. In other words we are enforcing the programmer to take care of the scale if he wants to multiply two temperatures. In all other cases the scale used by the programmer to assign the temperature of the objects is irrelevant.

On the contrary we provide two division operators. The operator that permits the division of a temperature by a number returns a temperature, while the one that allows the division of a temperature by another temperature returns a number.

We also define autoincrement and autodecrement operators, as well as an autodivision operator. The automultiplication operator `*=` is not defined for the reasons explained above: the result will not be neither a temperature, nor a number. In this way the programmer attempting to execute something like

```
t2*=t1;
```

where `t1` and `t2` are temperatures will get an error at compile time. Note that all these operators are meant to modify the leftmost operand, then they return an object of class `Temperature&`.

Laboratory 3++.1 *Understanding classes and objects.*

Write down the implementation of all the methods of the class `Temperature` and write a program to test all of them. Try using catenated operators, then rewrite the definition of the class in such a way that operators returning objects of class `Temperature&` returns objects of type `Temperature` and see what happens. Modify constructors and the destructor in such a way they always write down a message informing the user when they are called and the address of the object to be created or destructed (remember that this address is contained in the `this` attribute). Follows the execution of your test program and see if messages appears when you expect them. In particular, consider following the execution of the test program in both versions: the one in which operators return the address of the objects and the other in which they return objects. Pay attention to what happens for operators like `+`, `-`, `*` and `/`.

Chapter 5++

Working with batches of data

In this chapter we learn how to use data structure in C++. Of course all C structures can be used as well, however C++ provides much more flexible structures that can be very useful for scientists. In fact, these new structures are *added* to C++ by including a standard library, called STL (Standard Template Library). A template class is a class whose members belongs to a parametrized class. In C++, in fact, not only data, but also types can be parametrized as *variables*. This characteristics provides a further level of abstraction and polymorphism. In this chapter we show how to efficiently use STL objects and how to define new template classes.

5++.1 Vectors

Vectors are data structures that are much like an array. Vectors, however, are much more user friendly than arrays, since one can easily build vectors of any kind of objects and the programmer must not take care of its size.

In order to use vectors in you program or classes you must include the corresponding definition file:

```
#include <vector>
```

To build an object of the class `vector` you need to specify the class of the objects to store in each vector component. For example, a vector of integer numbers is declared by

```
std::vector<int> x;
```

Here `x` is a vector of integers. Of course you can declare vectors of more complex objects. For example, a vector of `Temperature` objects is declared by

```
std::vector<Temperature> t;
```

Data can be sequentially inserted into a vector using the `push_back` method. Listing **??** shows how to fill a vector and read back its content. First of all note that we prepended the string `std::` to the word `vector`: that means that both `vector` and `cout` belongs to the Standard Template Library.

Note also that, contrary to C, we are encouraged to declare objects just before using them. For this reason the variable `n` is declared after the first instruction, and the variable t i used in the first loop is declared inside it.

In the loop, we read integers from the keyboard and store them in subsequent components of the vector `x`. Note that we don't need to declare how long is the vector `x` prior to its declaration, nor while using it. All the memory management is done by the object itself. The space occupied by the data in the memory is increased, if needed, at runtime. The actual size of the object is given by the method `size()` who returns an integer counting the number of components.

In the second `for` loop we show how to read back stored values. The syntax is exactly the same used for arrays. The square brackets operator returns the i-th component of the vector. You can use this operator even to modify the content of a component such as in

```
x[i] = k;
```

However, you cannot use the assignment operator on a component, unless the component has been inserted in the vector. If you substitute the line with the `push_back` method with the above line, the result is a segmentation fault error. The reason being that the object must be created in the memory prior to access it, and this is left to the `push_back` method. Alternatively, you can use the `resize(n)` method. It inserts `n` objects in the vector, with their default status, as defined by their constructor.

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

main() {
  std::cout << "How many data ? ";
  int n;
  std::cin >> n;
  std::vector<int> x;
  for (int i = 0; i < n; i++) {
    std::cout << i << ": ";
    int k;
    std::cin >> k;
    x.push_back(k);
  }
  std::cout << "You inserted " << x.size()
            << " values in vector x" << std::endl;
  while (!x.empty()) {
    for (int i = 0; i < x.size(); i++) {
      std::cout << x[i] << std::endl;
    }
    x.pop_back();
  }
```

```
}
```

Listing 5++.1 Using a vector.

In the `while` loop we check if the vector contains still data. In fact, the `pop_back` method, removes the last inserted component and, repeating the instructions in the loop, the `x` vector will loose one component each time, until it becomes empty. You can test if a vector is empty or not using the `empty` method. It returns a value of type `bool`: this is a new type defined in C++ that represents logical or boolean values. It can assume two values: `true` or `false`. Of course you can continue using C definitions for true and false, however the C++ one is much more consistent.

Once the vector is filled, you can play with the `[]` operator exactly in the same way you use it for arrays. Many operations, however, are much simpler. For example, a vector can be copied into another vector just using the `=` operator:

```
std::vector y = x;
```

The first and the last element of a vector are returned by, respectively, the `front` and `back` methods, so `x.front()` is equivalent to `x[0]` and `x.back()` is equivalent to `x[x.size() - 1]`. The content of two vectors can be swapped using the `swap` method like

```
y.swap(x);
```

Note that `x` and `y` may have different size. To assign the same value to all the components of a vector at construction time you can use the constructor

```
std::vector<int> x(100, 0);
```

that builds a vector of 100 components, each of value 0. In this case you can start using the `[]` operator soon after the construction, since the objects already populate the vector. The `clear()]` method deletes all the vector components.

5++.2 Iterators

The `[]` operator is a good and familiar tool to navigate through objects like vectors, but is not always efficient and, moreover, is undefined for data structure that do not provide any order within the data, as in the vector.

C++ provides a much more efficient and generic way to navigate through data structures, by means of objects called *iterators*. An iterator is something that iterate over a data structure visiting all the components of it. Each class may have its own iterator. The second `for` loop in Listing 5++.1 can be rewritten using iterators as

```
std::vector<int>::const_iterator i = x.begin();
std::vector<int>::const_iterator e = x.end();
while (i != e) {
  std::cout << *i << std::endl;
  i++;
}
```

the first two lines defines two constant iterators. These iterators can be used to read data, but not to modify them. Non constant iterators can be constructed as std::vector<int>::iterator. The i iterator gets the value returned by the begin() method, i.e. it *points* to the first element of the vector. The method end() returns an iterator that points to the object past the last object in the vector.

Objects pointed by the iterator i is represented by *i: an iterator preceded by an asterisk operator is an object. An iterator pointing to an object in a data structure, moves to the next object once the ++ operator is applied to it. When i becomes equal to e it means it just passed the last object in the vector and the loop can be abandoned.

Using iterators it is very easy to perform operations that are extremely difficult using arrays: insertion and deletions of part of the data. Suppose you have a vector of 100 components and you want to add one more object k at position number 95. In C++ you can do it as easily as

```
x.insert(x.begin() + 95, k);
```

You can also insert a vector y into another vector x at the desired place:

```
x.insert(x.begin() + 95, y.begin(), y.end());
```

From the above example you can easily understand how to insert a portion of y into x: it is enough to compute the iterators to the first and the last object of the vector to be included in the target one.

Erasing objects from a vector is as easy as inserting; the erase method can be called with one or two iterators as parameters. In the first case only one element is removed from the vector, while in the latter case all elements between the two iterators disappear. Of course the size of the vector is changed accordingly, as well as the indexes of the elements.

It is worth noting that STL vectors are not at all mathematical like vectors. Their size, in fact, may change and no mathematical operators, such as addition, subtraction, multiplication and division by a scalar and the product of vectors are defined for them. They are intended as *containers* for objects that can be addressed by a numerical index.

5++.2.1 Namespaces

Multiple classes can be grouped into a *namespace*. A namespace contains one or more unique identifiers for classes, constants, symbols, etc.. Collecting identifiers into a namespace makes it possible to define other elements of the language with the same name of those contained into a namespace. The compiler distinguishes two elements with the same name by inspecting their namespace.

STL classes, constants and functions are contained into the std namespace. That's why we must prepend the std:: string to the name of the classes or objects to use them. The definition of STL elements into the std namespace, makes it possible to define our own vector class without ambiguity with respect to the one defined by STL.

Programmers can omit the namespace prefix if all the identifiers used in a project are unique. In this case the default namespaces must be specified. To avoid repeating the prefix `std::` each time you construct an STL object, it is enough to add, at the beginning of an instruction block delimited by braces the clause

```
using namespace std;
```

The clause will be valid along the entire block. Putting it at the beginning of the file (i.e. before the `main` or before class declaration, makes it valid for all the blocks in that file. From now on, we assume that the namespace for STL is used.

5++.2.2 Multidimensional arrays and vectors

Thanks to the fact that operators can be catenated, vectors with multiple indices can be defined as well, as for multidimensional arrays.

```
vector<vector<int> > x;
```

The above statement declares `x` as a vector of vectors of integers. Note the space between the `>` signs. This space is mandatory in order for the compiler to distinguish it from the `>` operator. You can use `x` indifferently as an object whose components are vectors, so you can do something like:

```
vector<int> y;
x.push_back(y);
```

or as an object whose components are integers addressed by two indices:

```
int k = x[10][20];
```

provided that the addressed objects exist. With respect to arrays, vectors are much simpler to use, especially compared to multidimensional arrays. You may remember that passing a multidimensional array to a function in C is a hell, while exchanging vectors between objects is as simple as exchanging a simple variable.

Again, there is no magic in this. The memory management is, in principle, as complex as in C. However, in C++ is the object itself that takes care of that. Objects *knows* where they are in memory, how much space they use, how it is filled and what to do if more space is required. All the complexity is encapsulated within the class definition.

Laboratory 5++.1 *Linear Algebra with vectors.*

Repeat the exercise described in Chapter 5 of the book, to solve a system of linear equations using the Gauss method. Define a class of type `myVector` that represents coordinate vectors and a class of type `matrix` for a square matrix. Use STL vectors to store the status of each object. Makes them independent on their size. Define operators for addition and subtraction of vectors, as well as operators for the multiplication and division of vectors by a scalar. Define an operator to multiply a matrix for a vector. Then write down the Gauss algorithm using objects of these classes.

5++.3 Strings

Strings in C are arrays of characters. STL defines objects of class `string` that behaves much like C strings, but have many advantages with respect to them. First of all, the assignment operator `=` is defined for strings, so string assignment is as simple as

```
string s = "This is a string";
```

No attention must be paid to the *length* of the string, nor to the fact that it must end with a null character. It is the object who takes care of this. String can be easily catenated using the `+` or the `+=` operators. All the operations defined for vectors are also defined for strings: insertion, deletion, etc.

The length of a string is returned by the `length()` method that takes into account the trailing null character, so that

```
string s = "Test";
cout << s.length() << endl;
```

returns 4, despite the fact that the real size of the string is 5 characters. The `[]` operator returns a character at the given position, acting just like an array of character (remember that the indices of the characters runs from 0 to `length()-1`).

Sometimes it is useful to treat the string like a C array. For example you may want to use C functions that accepts `char *` variables as parameters in your C++ program. In this case you can use the `c_str()` method that returns a pointer to an array of character, so you can do something like

```
string s = "I like C++";
printf("%s\n", s.c_str());
```

even if this is not a good idea, since I/O in C++ is better done using `iostream`.

5++.4 Lists

A list another type of STL container. It is much like a vector, i.e. it is intended to contain a list of objects, and has practically all the same behavior. However, while vectors are intended to organize data in such a way that the order in which they are stored is almost immutable and has a meaning for the programmer, lists are intended to store collections of data for which the order is irrelevant.

As an example, we use a vector `x` if we want to represent a set of coordinates in the space, since each index correspond to a coordinate, so `x[1]` has a different meaning with respect to those of `x[2]`. On the other hand, if we want just to collect data irrespective of the order, for example a set of repeated measurements of the same temperature of an object, we may want to use a list:

```
#include <list>
#include <Temperature.h>
...
  list<Temperature> t;
```

Populating the list is as easy as for vectors, using the push_back method. Lists can only be navigated with iterators, since no [] operator is defined for them, the reason being that the order of objects in the list is irrelevant and may change with the lifetime of the object, so that addressing t[i] may return an object or another according to what happened during the execution of the program.

Since ordering does not make sense for lists, they provide methods like unique(), sort(), reverse() and merge(). The unique() method removes all duplicated objects from the calling object. The list, then, may shrink. sort() changes the order in which objects are stored in the list. It can be called only if comparison operators are defined for objects in the list. In the case of temperatures we need to defined the following operators:

```
bool operator<(const Temperature &r);
bool operator==(const Temperature &r);
bool operator>(const Temperature &r);
```

Combined operators such as >= or != are automatically built composing the above mentioned methods.

Iterating from begin() tp end() a sorted list returns the objects in the ascending order. Using rbegin() and rend() as the initial and final iterator values, returns the objects in the opposite order.

Now reconsider programs written to order a batch of data and to find their maximum and minimum values, shown in Chapter 5. Since lists contains their own sorting algorithm, in order to obtain the same results, it is enough to store data into a list, sort it, and read back the first and the last element. Note that STL algorithms were written with emphasis on efficiency, so all the operations are the fastest possible.

The reverse() method, of course, just reverse the order in which objects are stored in the list, while merge() is used to merge two ordered lists into one ordered list:

```
list<Temperature> t1, t2;
...
t1.sort();
t2.sort();
t1.merge(t2);
```

makes t1 a list composed of ordered objects of both lists.

5++.5 Maps

Maps are associative containers. With maps you can store into a container a set of objects, each of which has a unique identifier. It is much like a vector, where objects are stored into a container and each of them can be addressed by a unique integer identifier, however maps provides the ability to associate to each object another object as a key. You can think to a map as a container for (*key, value*) pairs.

Maps are always sorted containers, and this makes retrieval of objects very fast. Maps uses binary search to find data in the container, based on the key. Since they are sorted containers, objects stored in a map must define comparison operators.

The key of a map can be an object of any class. Look at Listing 5++.2.

```
1  #include <Temperature>
2  #include <string>
3  #include <map>
4  #include <iostream>
5
6  using namespace std;
7
8  main() {
9    map<string, Temperature> tmap;
10
11   tmap["warm"] = 32.;
12   tmap["cold"] = 0.;
13   tmap["hot"]  = 100.;
14
15   map<string, Temperature>::const_iterator i = tmap.begin();
16   map<string, Temperature>::const_iterator e = tmap.end();
17   while (i != e) {
18     cout << "T = " << (*i).second.getC() << " C = "
19          << i->second.getF()
20          << " F label = " << i->first << endl;
21     i++;
22   }
23   cout << "The hottest Temperature is " << tmap["hot"].getC()
24        << endl;
25 }
```

Listing 5++.2 Using maps.

At line 9 we define an object of class `map`. The map stores objects of class `Temperature` and associates to each of it a key of the class `string`. Association is done in the lines following the declaration. An object whose temperature corresponds to 0°C is associated to the string `cold`. Note that this is possible because `tmap["cold"]` is an object of class Temperature and we defined an assignment operator that accepts a number as the right operand. Maps stores objects in it as pairs: another STL container. A pair is an object composed of two objects, whatever their class. In our case the map stores objects of class `pair<string, Temperature>`.

We iterate over the whole map using the iterator `i` to visit all its objects. Each time the iterator returns one of the elements of the map. The object *pointed* by an iterator is obtained applying the `*` operator to the iterator, so `(*i)` in line 18 is a pair. The objects composing a pairs are returned by `first` and `second`, respectively. Note that they are not methods, since they do not have parenthesis. They are, in fact, public members, so they can be accessed from outside of the class just as public methods. `(*i).second`, then, is an object of clas `Temperature` so we can use its `getC()` method to get its temperature in degrees Celsius.

In the following line we use a different operator to get the object pointed by an iterator. It's the arrow operator `->` that can be used instead of the dot operator on an iterator. It works exactly in the same way. In fact `i->second` is equivalent to `(*i).second`. In the latter case we had to use parenthesis to change the priority with which operators are applied: the asterisk operator is applied prior to the dot operator. The label of each temperature is obtained in the following line with `i->first` that in fact is equivalent to `(*i).first`.

Executing the code, you will see that data comes with keys in alphabetical order, irrespective of the order in which pairs are inserted in the map.

One can access directly one of the objects, passing its key as a parameter of the operator `[]`, as in the last line. Of course keys must be unique within a map. Using the `[]` operator makes maps very similar to vectors, but now the key can be any other object, not only an integer number. The retrieval on an object within the map is fast, because it uses the binary search to find the object in its collection.

Be careful using the `[]` operator! In fact, it returns a pair if the key is found in the map, but, if not found, it inserts in the map an object composed of a pair whose key is the one provided as a parameter and whose value is the default value for the second object of the pair. As an example, suppose we add the following lines at the end of our program in Listing **??**.

```
cout << tmap.size() << endl;
cout << tmap["invalid"].getC() << endl;
cout << tmap.size() << endl;
```

we should see something like

```
3
0
4
```

The first number is the current size of the map (3 as the number of objects we inserted in it). Soon after we ask the program to print the temperature of an object whose key is `invalid`, that does not exists in the map. The result is the default value of the temperature of an object of class `Temperature` as defined in its default constructor. What happened is that the object was not found in the map, then it was defined using the default constructor. When we show again the size of the map, then, we see 4, since it increased by 1 after the insertion of an object with key `invalid`. If you want to check that the object exists, better use the `find` method as

```
if (tmap.find("invalid") != tmap.end()) {
  cout << tmap["invalid"] << endl;
}
```

Since map is a container it also has the typical methods of containers: `insert`, `erase` and many other.

Laboratory 5++.2 *Counting words.*

Repeat the exercise worked out in Section 5.6.3 of the book, adding a feature consisting in counting the occurrence of each word. Use a map where the key is a string and the value is an integer. Each time you encounter a new words (you can test it using the `find` method), initialize the value to 1. Then, you you find again the same word, increase the value on the element of the map having the same key. Iterating over the map, print the key of each object and, close to it, the number of occurrences. Remember that a map is an associative sorted container.

5++.6 Other classes

STL provides lot of useful classes. Describing all of them, as well as deeply discuss all the methods of the classes given above, is out of the scope of this book. For a complete reference just type `STL C++` in a search engine like Google and find tons of pages describing the usage of each class.

5++.7 Defining new template classes

STL classes are so versatile that hardly you will need to define new template classes. Despite this fact, it can always happens that you need new template classes to solve your specific problem. Moreover, the spirit of this book is unchanged moving from C to C++: we are learning how to use an instrument as scientists. To learn how to use a multimeter or an oscilloscope, from the practical point of view, you can ignore completely how it works, but not if you are a scientist. In this case you are requested to know what happens inside the instrument, in order to use it consciously and even in uncommon ways. The same applies for programming languages: you can completely ignore the internal details of the language, but not if you are a scientist. A programming language is an instrument and as such you have to know how it works to profit of it.

A template class is a class where the class of objects manipulated by the class is given as a parameter. In procedural programming we would say that types becomes variables.

Suppose, for example, that we want to write a class intended to hold experimental data collected in an experiment. Each measurement y is collected at some coordinate x, so $y = y(x)$. The class shall be able to provide values coming from statistical computations, such as the average value $\langle y \rangle$ of the measurements. Each data point y_i has an index i ranging from 0 to $N-1$, where N is the number of measurements.

The average is computed from the usual formula:

$$\langle y \rangle = \frac{1}{N} \sum_{0}^{N-1} y_i \,.$$

Note that we made no statement about the nature of y. It can be a scalar, as well as a vector in an n-dimensional space or a complex number. Whatever its nature, the above formula holds, provided the sum of two objects is defined, as well as the division by a scalar. The same holds for x. The independent variable can be a time, a point in the space, a coordinate along a line, etc.. In other words, the *type* of both x and y is not specified and can be *parametrized*. A possible parametrization is given in Listing 5++.3.

```
#ifndef _DATA_H_
#define _DATA_H_

#include <vector>
#include <bits/stl_pair.h>

template <class Tx, class Ty>
class Data {
 public:
  typedef typename std::vector<Ty>::const_iterator
          const_ty_iterator;
  Data();
  void push_back(const Tx& x, const Ty& y);
  Ty average();
  std::pair<Tx, Ty> find(int i);
  std::pair<Tx, Ty> operator[](int i);
 private:
  std::vector<Tx> _x;
  std::vector<Ty> _y;
};

template <class Tx, class Ty> Data<Tx, Ty>::Data() {
  _x.clear();
  _y.clear();
}

template <class Tx, class Ty> void
  Data<Tx, Ty>::push_back(const Tx& x, const Ty& y) {
```

```
  _x.push_back(x);
  _y.push_back(y);
}

template <class Tx, class Ty> Ty Data<Tx, Ty>::average() {
  const_ty_iterator i = _y.begin();
  const_ty_iterator e = _y.end();
  Ty m = 0;
  while (i != e) {
    m += *i++;
  }
  return m/_y.size();
}

template <class Tx, class Ty> std::pair<Tx, Ty>
  Data<Tx, Ty>::find(int i) {
  std::pair<Tx, Ty> r;
  r.first  = _x[i];
  r.second = _y[i];
  return r;
}

template <class Tx, class Ty> std::pair<Tx, Ty>
  Data<Tx, Ty>::operator[](int i) {
  return find(i);
}

#endif
```

Listing 5++.3 Defining a template class.

First of all notice that we put everything (declaration and definition) into the same `Data.h` file. This is mandatory since template classes cannot be compiled independently from the files in which they are used. In fact, the compiler must know the types of the objects to manipulate to correctly produce the machine code and this can only be obtained once an object of the template class is instantiated.

We chose to represent x and y as two vectors, rather than a vector of pairs. This will simplify data manipulation. Since we use vectors we included `vector` in the file. A single measurement, however, is a pair (x, y), so we need pairs; that's why we included `bits/stl_pair.h`.

Since we do not know the class of objects manipulated by the class we define `Data` as a template class, declaring two symbols to represent the actual class of the objects: `Tx` and `Ty`. They are declared prior to the class definition using the syntax

```
template <class Tx, class Ty>
```

as in Line 7. Inside the class declaration we use Ty as the class specifier of an object of class Ty. For example, the method average() returns an object of class Ty, i.e. of the same class of objects representing data. The same holds for rms(). We provide a push_back() method to insert data in the object. Since each measurement is composed of a pair (x, y), this method has two arguments: x and y, of class Tx and Ty, respectively. The method find, equivalent to the operator [], returns a pair (x, y) as an STL pair composed of an object of class Tx and an object of class Ty. Attributes are organized as STL vectors of the corresponding classes. You can see how we treat symbols for unknown classes as classes.

Let's analyze the implementation of methods. The push_back method just execute the method with the same name for each vector used to store the data. Note that in the definition of methods we need to prepend the return type by the declaration of template symbols, while the class to which belongs a method (the string before ::) contains the names of those symbols.

The average method is very interesting: we defined two iterators i and e, as usual, to navigate along the vector representing y data. We defined a type using the typedef statement in the class declaration, in order to make the code more readable. The typedef statement declares a synonym for a type. In this case the word const_ty_iterator is a synonym for std::vector<Ty>::const_iterator. Note that in this case we needed to add the typename qualifier to specify to the compiler that what follows is a type name. This is needed because what's follows the :: operator is a *dependent name*, i.e. a name whose meaning depends on some argument: in this case on the actual class name used for Ty.

The average method returns an object of the same class to which y_i belong, represented by Ty. To compute the average we need to initialize the object to some conventional 0 value (then, we must be sure that the = operator is defined for the actual class and that assigning the value 0 to it is allowed). For example, if the object is a scalar this is trivial, but for vectors $\vec{x}$ in space we may need to define an operation such that $\vec{x} = 0$ means that $\vec{x} = (0, 0, 0)$. Then we loop over all y_i and compute the sum of all y_i. To do that we need to be sure that the operator += is defined for objects of class Ty. Note how to use an iterator and to move it to the next element at the same time. At the end of the loop we return the sum divided by the number of measurements. Again, this makes it mandatory to define the operator representing the division by a scalar of objects of class Ty.

Using this class is very simple. Look at Listing 5++.4. You need to include the file containing the template class definition and declares objects of type Data with the types of the objects representing x and y as parameters within <...>.

```
#include <Data.h>
#include <iostream>
#include <stdlib.h>

```

```
5 main() {
6   Data<double, int> d;
7   for (int i = 0; i < 100; i++) {
8     d.push_back(rand()/RAND_MAX, rand());
9   }
10  for (int i = 0; i < 100; i++) {
11    std::cout << d[i].second << std::endl;
12  }
13  std::cout << "Average = " << d.average() << std::endl;
14 }
```

Listing 5++.4 Using template classes.

In this example we measure some integer data y varying some observable x, represented as a real number. For simplicity we generated data randomly. In this case the average is an integer. However, nothing prevents us to use objects of class `double` as well for y to represent scalar quantities, nor objects of our own class `Vector` to represent points in space. The main program, as well as the definition of `Data` do not change, apart from the detail in Line 6.

Chapter 6++

Pointers in C++

In this chapter we see how to use pointers and associated operators in C++. Iterators are reviewed in terms of pointers and we learn how to create objects in the memory at runtime. Dynamic memory allocation is an advanced topic for C programmers; in C++ this topic is simple enough to be introduced at this stage. Of course, there is no magic in it. Object Oriented Programming, in fact, encapsulates complexity in classes, so what was difficult in C becomes relatively easy in C++, provided we hide the details of the class implementation. Pointers in C++ are much less used than in C, since most of the operations that require a pointer in C, require an object in C++. However, the knowledge of pointer's usage and arithmetics is mandatory for good and efficient programming.

I/O is revised, too. Writing and reading data to and from files, respectively, is done via objects in C++. We learn how to describe the way in which an object can be *written* or *read*.

6++.1 Pointers

Pointers in C++ works just as in C. You can declare a pointer to any object using the asterisk after the class name; for example:

```
Temperature *t;
```

declares a pointer to a Temperature object. The pointer `t`, as usual, stores the address of an object of class `Temperature`. Then `t` is represented as a sequence of n bits, where n depends on the machine architecture, that represents an address in the memory.

Pointers can be assigned, as in C, using the `=` operator, if the right hand operand is an address as well, as in

```
Temperature t1 = 100;
Temperature *t = &t1;
```

The `&` operator being the reference operator, returning the address of its operand. Pointers arithmetic is exactly the same it was in C. Summing an integer k to a pointer p, $p+k$

makes it point to the object in the memory located $k \times m$ bytes far from p, i.e. the new address p' is in fact $p' = p + k \times m$.

Accessing methods of an object through a pointer requires the pointer to be dereferenced, as in C, like `(*t).setC(200)`. Alternatively you can use the *arrow operator* `->` to access the method directly via the pointer, as in `t->setC(200)`.

6++.2 Dynamical memory allocation

As in C, in C++ programming, objects must be declared in the program before being used, in order for the compiler to set up the machine instructions needed to allocate the necessary amount of space in the computer memory, to store objects' status. In some cases, however, we may want to be able to create objects at run time, i.e. to allocate memory dynamically. Using pointers to objects is also useful to profit from another important property of OO programming: polymorphism, discussed in Chapter ??.

Dynamic memory allocation is done using the `new` operator, like in the following code fragment:

```
Temperature *t = new Temperature;
t->setF(200);
std::cout << t->getC() << std::endl;
```

Here `t` is a pointer to an object of class `Temperature`. No such object has been declared in the program, so we cannot assign its value using the reference operator. Moreover, at the time the program starts, there is no space allocated in the memory for an object of class `Temperature`. Such a space is requested to the operating system by the program, at runtime, once the `new` operator is applied. The first line of code tells the system to allocate enough space for an object of class `Temperature` and returns the address of the corresponding memory chunk. This adddress is assigned to `t` that can now be used to perform any operation allowed for it, using the resolution operator `->`.

Once object are declared in a C++ program, their constructor is called. The same happens when an object is dynamically created with the `new` operator. In this case only the default constructor (the one without parameters) can be called. As soon as an object goes out of scope it is destructed by the destructor. Its memory is released too. The compiler takes care of this. The same must happen once you create objects dynamically, but in this case you must take care of objects' disposal, since the compiler cannot know how many objects are going to be created in the memory at compilation time. To release the memory occupied by an object and call its destructor, you need to apply the `delete` operator to it:

```
delete t;
```

If you are working with arrays of pointers, you can destroy all the objects in a dynamically created array adding a couple of square brackets between the `delete` operator and the array name. For example, in the following piece of code

```
int n;
std::cin >> n;
Temperature *t = new Temperature[n];
for (int i = 0; i < n; i++) {
  double tf;
  std::cin >> tf;
  t[i]->setF(tf);
}
...
delete [] t;
```

we allocate an array of `n` objects of class `Temperature`. Note that the size of the array is given at run time: it is obtained as an input value from the user. Objects in the array are accessed via the `[]` operator; since each objects is a pointer, methods on them are called via the arrow operator. To delete all objects we use the `delete [] t;` instruction, telling the system to destruct all objects stored in the `t` array before releasing the memory.

6++.3 Implicit references

STILL TO BE WRITTEN

6++.4 Object I/O

An object can be quite complex to be represented on a screen. For example, consider an object of class `SquareMatrix` that represents a square matrix. Its status can be represented either by a two–dimensional array as well as a more complex structure. In the laboratories of Chapter 5++ we suggested to represent a matrix as an object composed of rows of objects of the class `myVector`. The status of a `myVector` is represented as a `vector` of `double`, while the matrix is represented as a `vector` of `myVector`. Defining operators `[]` for both, we can access the elements of the matrix and of the vectors as `A[i][j]` and `v[i]`, where `A` and `v` are the identifiers of the matrix and the vector, respectively.

If we want to show the status of a $n \times n$ matrix on a screen we should write some piece of code to arrange the elements in n rows of n columns. We may also want to add some character grouping the data. Consider, for example, the following matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

To represent it on a computer screen we can write it as

```
| 1  2  3 |
| 4  5  6 |
```

```
| 7  8  9 |
```

To do that we need to loop over rows and columns, write the corresponding element content, and add a `|` character at the beginning and at the end of each line, taking care of newlines. It would much more easy and safe to be able to do something like

```
std::cout << A << std::endl;
```

It turns out that you can, in fact. It is enough to define how to perform the operation, defining the proper insertion operator as a function. The new insertion operator must be defined as a function and not as a method of `std::cout` since you have no control of it. You cannot change its behavior. You can, however, define a function that takes two arguments: the *output stream* and the object to be written into it. When the computer executes the instruction given above it executes the function using as output stream the `std::cout` object.

A possible definition of the function is given in Listing 6++.1.

```
#include <Matrix.h>

std::ostream& operator<<(std::ostream &o, const Matrix &r) {
  int s = r.size();
  for (int i = 0; i < s; i++) {
    o << "| ";
    for (int j = 0; j < s; j++) {
      o << r[i][j] << " ";
    }
    o << "|" << std::endl;
  }
  return o;
}
```

Listing 6++.1 An insertion operator for matrices.

The return type of this function is always `std::ostream&` because insertion operators can be catenated. The parameters used by the functions are `o`, passed to the function via its address, and `r`, an object of type `t`, that of course must not be changed during the execution of this function and is declared to be `const`, despite the fact we pass its address to the function by means of the reference operator.

As soon as the compiler encounters a line like

```
std::cout << A << std::endl;
```

where `A` is a `Matrix`, calls the above mentioned function, where `o` is substituted by `std::cout` and `r` by `A`. What happens is then that we start a loop on rows, in which we send to the output stream the character `|`, then the matrix elements on that row separated by one space, followed by another `|` character and a newline.

At the end, the output stream is returned, so that `std::cout << A` is in turn the output stream itself, thanks to the return type `std::ostream&`. The last newline character is sent to the latter. The result is what we got above. In this case, since we do not access private or protected members of the `Matrix` in the function, we don't need to declare the function elsewhere. However, it is a good practice to declare the function within the corresponding class.

As a rule, we would declare the insertion operator function in the `public` section of the class for which we are writing the function, prepending the keyword `friend` to it, e.g. in the `Matrix.h` file. We then write its implementation code in the definition file, e.g. in `Matrix.cc`. Declaring a function as a `friend` of a class, makes it possible for the function to access directly member data, without the need to be mediated by methods. This can be useful in those cases in which efficiency is an issue and in those cases in which no useful methods are provided to write the status of the object.

Note that we define the operator as a function that accepts a generic output stream as a parameter, irrespective of its concrete type. `std::cout` is an output stream, but it is not the only one. Any file open for writing is in fact an output stream. To open an output stream as a file, include the `fstream` file and instantiate an object of type `ofstream`. The name of the file can be passed to the constructor as a parameter, like in

```
ofstream f("/home/user/test.dat");
f << A << std::endl;
f.close();
```

where we declare a stream `f` representing a text file in `/home/user/test.dat`, to which we send the content of the `A` object as given by the insertion operator. Then, we close the stream.

Of course you can also write a function for input. Once you defined how the input is provided, you just need to define the proper extraction operator, in the `public` sector the the `Matrix` class, as

```
friend std::istream& operator>>(std::istream &o, Matrix &r);
```

Note that in this case the right hand operand is not constant, since it is going to change after the input is terminated.

In the implementation file you then write the code for the operator. We define the matrix elements to be passed from the stream as a set of characters starting with the character `(` and ending with `)`. Elements are separated with a comma. The code for the extraction operator is in Listing 6++.2.

```
std::istream& operator>>(std::istream &input, Matrix &r) {
  std::string s;
  char c = 0;
  while (c != ')') {
    input >> c;
    if ((c != ' ') && (c != '(') && (c != ')')) {
```

```
7         s += c;
8       }
9     }
10    int i = 0;
11    int j = 0;
12    int l = s.length();
13    int e = s.find(",");
14    while ((e > 0) && (e < l)) {
15      std::string v = s.substr(0, e);
16      std::istringstream buffer(v);
17      buffer >> r[i][j++];
18      e = s.find(",");
19      s = s.substr(e + 1);
20      if (j == r.size()) {
21        i++;
22        j = 0;
23      }
24    }
25    return input;
26  }
```

Listing 6++.2 An extraction operator for matrices.

In this code we need to read a string from the stream, ignore the first and the last characters, split the string on commas and transform each substring into a number. Lines between 2 and 9 are used to read the string from the stream. The result is put into `s`. Note that we read a single character at each step and discard blanks and the parenthesis. We stop reading once we encounter a `)`[1]. Once we have the whole string we use the `string` methods to

1. extract a substring from `s` using the first n characters, where n is the position of the first `,` character (line 15);
2. transform the latter into a number by means of an objects of class `istringstream` (an input string stream), whose constructor accepts a string as a parameter and whose extraction operator returns a properly formatted value (line 16);
3. using the extraction operator of the string stream we insert the corresponding numerical value of the last substring into the proper matrix element (line 17);
4. we remove used characters redefining `s` as the part of string after the first comma (line 19);
5. in the following loop we update the row index, once we have read a number of elements equal to the columns of the matrix.

This process is iterated until the input string is over. Suppose that now we have a text file `matrix.dat` whose content is

```
(1, 2, 3,
```

[1]The code presented here is not very robust, since it does not check the expected format. However this is beyond the aim of this section.

```
4, 5, 6,
7, 8, 9)
```

We can instantiates an input stream representing this file and extract data from it into a `Matrix` object `A` as

```
ifstream f("matrix.dat");
f >> A;
f.close();
```

Of course the formatting is completely irrelevant. We can even input data from the keyboard as

```
(1,2,3,4,5,  6,      7,      8,9)
```

Note the blank characters. We can add them freely into the input string, since they are ignored by the extraction operator.

Laboratory 6++.1 *Output formatting*

Rewrite the insertion operator for a matrix in such a way that the matrix elements are nicely aligned. A possible way to do that is to write all the elements in scientific notation, with a fixed number of digits. This can be achieved using the `setf` method of the output stream, that takes an integer argument. The argument is a collection of bits that, if set, activates a feature. Formatting bits are defined into the `ios` namespace of the `std` library. To activate the scientific notation, it is enough to call `o.setf(std::ios::scientific)` before using the stream `o`. Remember to unset the bit with `o.unsetf(std::ios::scientific)`. There are many interesting formatting bits in the standard library. Search websites for more. You can also set the precision of the number you print inserting a modifier into the stream. Modifiers are returned by appropriate functions, defined into the `std` library. To limit the number of digits after the decimal point to 2, for example, you can insert into the stream the result of `setprecision(2)` like in

```
o << setprecision(2) << r[i][j] << " ";
```

Another possibile way to nicely write matrix elements is to use a fixed number of digits for the integer and decimal part of its value. To compute the appropriate number of digits, you need to know how many characters are needed to write the longest number. Let's call it m. To align elements on the decimal point you need to know the number of digits of the integer part of the largest; let's call it n. To get these numbers you can play with logarithms and some arithmetics, as well as with string streams to transform numbers into strings. Each element must be represented using at most $m + 1$ characters (the extra characters being blanks, we do not need to add it after each number). Moreover, it must have n characters before the decimal point. The instruction to force the stream to write numbers in this format is

```
o << setprecision(m - n - 1) << setw(m + 1) << r[i][j];
```

The `setw` function tells how many characters must be used to write each element. The precision tells how many digits to write after the decimal point. Since the largest number is composed by m characters, taking into account one character for the decimal point, the number of digits after the integer part must be $m - n - 1$.

Chapter 7++

Functions vs Objects

Traditional programming languages have to deal with data, manipulated by functions. On the contrary, Object Oriented Programming has to deal with objects and messages exchanged between them. In this chapter we discuss this point in detail. This is not a technical chapter, but it is very important. In fact, most of those who learned a traditional programming language like C, tend to write C++ programs like C revised, and there is no real advantage in this. In order to be a good object oriented programmer, you must abandon the way of thinking in terms of data and functions. You must force yourself to think in the same way you think when you write down the solution of a problem using a pencil and a sheet of paper, in fact. Even if most programmers do not realize it, solutions written in terms of formula and described in natural language, appear to be much different when turned into data and functions. Object oriented programming makes the formal description of solutions much closer to the natural language.

TO BE FINISHED

Chapter 8++

Inheritance and Polymorphism

In previous chapter we discuss how C++ makes computations much more easier with respect to traditional programming techniques. The ability to redefine operators, in particular, can be very appreciated by most of scientists and in particular by physicists.

C++, as most OO languages, can provide much more interesting features, like inheritance and polymorphism. Both described in this chapter introducing a numerical integration example.

8++.1 A numerical integrator object

In Chapter 8 of the textbook we show how to numerically compute the integral of a function. We discuss many methods: from a deterministic one, like the midpoint method, to a stochastic one, like the Monte Carlo method. In this chapter we discuss numerical integration in C++. Remember that in this language objects do the job. Of course a function to be integration is just a function: a C function. However the integration is done by an object: we could call it an integral or an integrator. An integrator takes at least a function and the two integration extrema, to perform its job. According to the method used, once can provide more parameters like the number of intervals in which the integration interval must be divided, the number of extractions for Monte Carlo methods, a fit model to extrapolate to the infinite number of divisions result, and so on. Let's start defining which is the main behavior of the method, leaving implementation details apart. Have a look to Listing 8++.1

```
#ifndef _INTEGRATOR_H_
#define _INTEGRATOR_H_

class integrator {
 public:
  integrator();

```

```
  integrator& setF(double (*f)(double));
  double f(double x);
  double integrate(double a, double b);
 private:
  double (*_f)(double);
};

#endif
```

Listing 8++.1 The interface of an integrator.

Beside the needed default constructor, we provide a method to let the integrator to know the function to be integrated (`setF`). The function, being generic, is represented as a pointer to a function. Then an integrator is capable of returning the value of the function to be integrated at a given point x (`f`) and the value of its integral between two points a and b (`integrate`). Note that in this case we are focusing our attention to the design of an integrator class and we want to get rid of other complications, but in general you can define the function $f(x)$ as a multidimensional function, letting x being a vector of something. The integrator can then be a template class and be able to compute the integral of very exotic objects.

Of course the main job is done into the `integral` method, that varies according to the chosen integration method. For example, for Monte Carlo it would be like the one reported in Listing 8++.2.

```
double integrator::integrate(double a, double b) {
  double S = 0.;
  for (int i = 0; i < _np; i++) {
    double x = (b - a)*rand()/RAND_MAX + a;
    S += f(x);
  }
  return S*(b - a)/_np;
}
```

Listing 8++.2 Monte Carlo integration.

Note that, to perform integration, the method needs an integer value, called `_np`, representing the number of random points to extract. From the leading underscore you can imagine that we defined it as an object member, to be added to `_f`. Of course we then needed another method to set it, and we may assign it a default value in the constructor.

On the other hand we may want to provide a deterministic integrator, whose interface is much like the same, where the integer parameter represents the number of subintervals in which $[a, b]$ must be divided. That means that we have to provide more than one class: one per integration method, all having more or less the same interface, then repeating the same code many times. This is annoying and, moreover, error prone: if we update a method in one of the classes, we may forget to do the same on the others, or we may mistype something in such a way that the code is still valid (it can be compiled), but wrong.

8++.2 Inheritance

The solution is *inheritance*: the ability for an Object Oriented Programming language, to define new objects starting from existing objects. New objects *inherits* the behavior of their parents. In fact, an object of class B, defined from an object of class A, *is* an object of class A, too, just as a car equipped with a GPS navigation system and automatic transmission is still a car. We just added some feature (the GPS navigation system or the automatic transmission) to an already working object (the car). In OOP we can define a basic integrator method, providing just the common interface to all integration methods. Starting from this we can then define specialized objects, each inheriting the basic behavior, but each providing a different way to compute the integral.

The integrator object contains the implementation of just its empty default constructor, an empty `integrate` (returning 0, just to let the compiler go) method, the `setF` method and the `f` method, the latter just returning `_f(x)`. We do not provide any generic `setN` method here, since the number and the type of parameters for each integration method may vary.

We then define another class, named `mcIntegrator`, that *is* an integrator in the sense that it is able to provide exactly the same behavior, plus some more. To define the `mcIntegrator` as an `integrator` by inheritance, we just provide this information to the class as in Listing 8++.3.

```
#ifndef _MCINTEGRATOR_H
#define _MCINTEGRATOR_H

#include <integrator.h>

class mcIntegrator : public integrator {
 public:
  mcIntegrator();
  mcIntegrator& setNpoints(int n);

  double integrate(double a, double b);
 private:
  int _np;
};
#endif
```

Listing 8++.3 A specialized Monte Carlo integrator.

We tell the compiler that `mcIntegrator` is an `integrator` by means of the line

```
class mcIntegrator : public integrator
```

meaning that what is public for an integrator is also public for a Monte Carlo integrator, while what was private for an integrator is still private for a Monte Carlo integrator. We included the `integrator.h` We added a new method `setNpoints`, intended to define the number of random numbers to extract to compute the integral, and we repeated the prototype of the `integrate` method, since we want to change its behavior with respect to the parent class. Note that we did not repeated the prototypes of the real common methods like `setF` or `f`. They just behaves as in the parent.

The content of the `integrate` method is, of course, the one shown in Listing 8++.2, the only difference being the name of the class in front of the double colon: `mcIntegrator` instead of `integrator`.

Once you defined `mcIntegrator` as above, you can use it in your application and you can call either methods specific to `mcIntegrator`, like `setNpoints`, or methods common to both parent and child, like `setF`. If you call a method common to both, redefined in the child class, then the latter is executed. You can, for example, do something like

```
mcIntegrator I;
I.setF(myFunction);
I.setNdivisions(8192);
I.integrate(0., 3.);
```

to integrate the function `myFunction` between $x = 0$ and $x = 3$ using 8192 random points. Cool, in fact!

Laboratory 8++.1 *Comparing integration methods.*

Write down the definition of at least three more classes of integrators. For each class find and implement the appropriate methods. Then write an application that, using those classes, computes the same integral and compares the results. If you know the primitive of the function, provide a method to the base class to return its value and use it to compare the estimated value of the integral with the real one.

What happens if you do not include the `f` method into the base class and use `_f` into the `integrate` method to perform the computation? Apparently this seems to be more appropriate, since `_f` is a member of the class and using it does not involve calling a method, making the execution slower! The compiler complains because `_f` is in fact a member of derived classes, but it is a private member of the base class, so it cannot be used as such in derived classes. The use of the method is then mandatory to permit access to it via the base class. If you want to avoid calling methods of a base class to access members, you can put them within the `protected` context of the base class, instead of the `private` one. This keeps members private for all classes, but for the derived ones. Do not abuse of the `protected` context! Use it only if needed. It weaken encapsulation.

8++.3 Polymorphism

Inheritance is cool, but *polymorphism* is even cooler, in fact! Polymorphism is the ability for an object to behave differently according to the context. The kind of polymorphism described in this section is in fact what is called as such by most textbooks on Object Oriented Programming.

Consider the example worked out in Laboratory 8++.1. Suppose you instantiated n different objects of m different classes. To compare results you can, of course, put each group of objects belonging to the same class into a list, loop on its elements and call the `iterate` method for each. But you cannot mix objects of different classes into a single list. Polymorphism allows you to do so. You can insert into the same list objects (or, more precisely, pointers to objects) of different classes, provided they have some common ancestor. Polymorphism works only with pointers, as explained in Section **??**, so to be able to profit from it we need to instantiates objects dynamically as pointers. For example:

```
list<integrator *> I;
for (int i = 1; i < 11; i++) {
  midpointIntegrator *mI = new midpointIntegrator;
  mI->setF(f);
  mI->setNdivisions(i);
  mcIntegrator *mcI = new mcIntegrator;
  mcI->setF(f);
  mcI->setNpoints(i * 1000);
  I.push_back(mI);
  I.push_back(mcI);
}
```

puts a total of twenty pointers to objects into a list called `I`. Ten of them belong to the class `mcIntegrator`, while the rest belong to the class `midpointIntegrator`. Each object has been configured using its own methods: Monte Carlo integrators are configured to compute the integral with an increasing number of extractions; deterministic integrators compute the integral dividing the whole interval into an increasing number of sub–intervals. Note that, despite the fact that some object belongs to a class and some other to another class, we can mix their pointers in a single list of pointers to their common base class.

The *magic* comes now. Impressively enough, if we loop on the elements of the list and call the `integrate` method for all, each object calls in fact its own integration method, despite the fact they are defined all as pointers to the same object `integrator*`:

```
list<integrator *>::const_iterator ii = I.begin();
list<integrator *>::const_iterator ie = I.end();
while (ii != ie) {
  cout << (*ii)->integrate(0., 10.) << endl;
  ii++;
}
```

It is like each object knows to which class it belongs and determines the appropriate method to be called at runtime. In order for the example to work as expected you only need to add the keyword `virtual` to the `integrate` method of the base class, as in

```
virtual double integrate(double a, double b);
```

If you want to make it impossible to instantiate objects of the base class `integrator` (remember, it returns zero as the integral of any function, and is then useless) it's enough to declare the method as *pure virtual* putting it equal to zero:

```
virtual double integrate(double a, double b) = 0;
```

so you don't even need to write its implementation in the `integrator.cc` file. Any attempt to instantiate an object of class `integrator` will fail at compilation time. Isn't polymorphism really cool?

Of course, even in this case, there is no magic at all. Everything works because the system maintains in memory not only the addresses of the data representing the members of a class, but even the addresses of their virtual functions, so they can be determined at runtime.

8++.4 How polymorphism works

Chapter 9++
Introducing UML

UML stands for *Unified Modeling Language* and is a powerful tool to describe an Object Oriented software project using a graphical language. UML is a language for software design and documentation, but it is a useful tool for preliminary testing, prior to the implementation phase.

A good design is the key of Object Oriented software. Starting from a careful design phase makes software development easy and easily maintainable. It is a common mistake to start writing code as soon as one has got some idea about a possible solution of a given problem. Object Oriented software, being suitable for large complex projects, requires some care since the beginning, to avoid the need to rewrite everything after discovering that the chosen design is not suitable for the given problem. If you want to build a dog's bed, you probably just write down a rather simple sketch of it on a sheet of paper, evaluate the length and the number of the boards, buy them and some stud, and start hammering. However, if you want to build a rather more complex building such as a small house, you need a much more detailed project. Such a project usually describes many aspects of the building at different scales: it shows views of the final result (useful for the owner to realize what is going to buy), a view of the supporting structure, completed with some technical information like the size of the pillars and the type of concrete to be used (useful for the builders), a schematic view of the services like water and gas pipes, and cables (to be used by other professionals and for future reference, in case of damages), etc. All the drawings are made using a conventional *language*: the size and the type of the lines used in the project have a very precise meaning. All engineers use the same convention, in such a way that all the professionals involved in the construction, as well as the customers, are able to decode the meaning of the drawings.

Mostly the same happens in Object Oriented software design. A good programmer starts with a description of the requirements of the customer, using a conventional language so that, if the project is large, its realization can be shared by many people who share the same knowledge of the description language. This part of the process is known as the *Use Case* study.

Then the analyst starts choosing among possible solutions and make drawings representing, at different levels of details, the involved classes and their relationships, i.e. it writes one or more *Class Diagrams*. Just like in the project of an apartment, there can be different views of the same element, at different scales. For example, in the drawing of the layout of the apartment, the engineer just shows the size and the position of walls, windows and doors, without many details. In other drawings the engineer may want to show the location of services. For example, the bathroom can be represented in a separate sheet with the location of the pipes and the services like the tub, the washbasin, etc. Each drawing is intended for different purposes, and in much the same way, the software analyst may want to describe its project at different scales: sometime showing just the relationships between classes, sometime illustrating the detailed content of classes.

Sometime a dynamic description of the project is needed, such as when you need to show the behavior of the interaction between two or more classes during time. In this case you need to prepare a *Sequence Diagram*.

Finally, since each component of the project can be installed on different nodes or components, you may need to specify the location of each component on a *Deployment Diagram*.

Each of these diagrams is to be realized in a common *language*, so that everybody can read and understand your project. Such a language is UML. A detailed description of UML is beyond the aim of this textbook. We will just describe some basic feature of the language and you can refer to the bibliography to find more information.

9++.1 Integration, revised

As usual we introduce new concepts through examples. Numerical integration is the chosen example. First of all, let's *describe* the problem, in terms of *natural* language: we aim to realize an application that, given a function $f(x)$ and two numbers $a, b \in \mathcal{R}$, is able to compute

$$I = \int_a^b f(x)dx\,,$$

with the desired degree of approximation. The numerical integration method may vary, depending on the type of the function $f(x)$, the integration extrema, the desired precision and the available resources in terms of CPU time. In the end, a user asks to the application to provide the value for I.

9++.1.1 A Use Case Diagram

A Use Case captures the intended behavior of a system: it describes what a system does, irrespective of how it is done. The problem of numerical integration is represented as an UML Use Case in Figure 9++.1

In this diagram we can recognize an *Actor*, represented as a sticky man, interacting with the system to realize the use case, represented by the ellipse. An actor is anything interacting with the system from outside: it can be a human, another software application

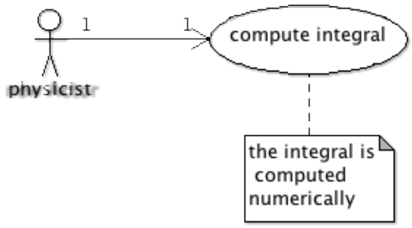

Figure 9++.1 A Use Case Diagram.

or a device. The *association relationship*, represented by an arrow, joining the actor with the use case, shows that the actor asks for the given use case to the system. The relationship may be named or not, depending on the needs. Notes can be added to any element of the diagram as text written inside a box with a dog–eared corner, connected with a dashed line to the element to which it refers. Other Use Cases can be much more complex, showing many relationships between different actors and systems.

Some elements of the language are common to many diagrams. As an example, notes can be used in all diagrams. In the next section we show other elements that can be used in Use Case diagrams, too.

9++.1.2 A Class Diagram

Once the analysis of the requirements has been done with the Use Case diagram, we start the most important phase of the design: we draw the *Class Diagram*. This diagram represents the classes needed to solve the given problem and is the basis for coding. In fact, a very detailed class diagram can be used to even *generate* automatically C++ code. It gives a static view of the system and sets the vocabulary used during the development phase.

After some thoughts, we already have an idea about how to realize the Use Case in terms of classes. We provide an abstract `Integrator` class, from which we derive many different concrete classes. Each of them provide the solution in terms of a different numerical integration method. However, their behavior is the same and is described by

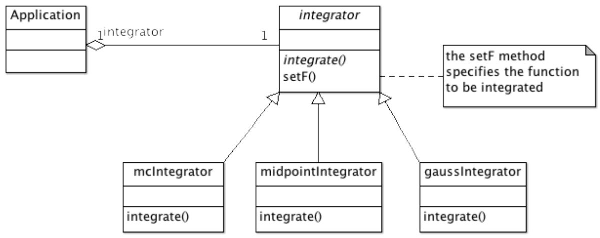

Figure 9++.2 A Class Diagram.

the base abstract class.

The class is part of an application that may take an object of an available concrete class as a parameter, to be used to perform the integration. The corresponding class diagram appears as in Figure 9++.2.

In this diagram, each class is represented as a rectangle divided into three parts: the top section contains the name of the class, the middle section the list of operations (methods) and the bottom section its attributes. There is no need to specify every detail: we just report the relevant operations and attributes.

The name of the class appear in italics if the class is abstract, i.e. it has at least one pure virtual method. Pure virtual methods appear in italics, too. Concrete classes appears much the same, but their name and methods appears in plain text. In Figure 9++.2 the `Integrator` class is abstract, since its `integrate` method is abstract. We specified that this class has a `setF` method, whose meaning is explained in details in the note.

Derived classes are said to be in a *"is a" relationship* with their base class and this kind of relationship is represented by an arrow connecting the derived classes to their base class. The arrow has the form of a triangle.

With respect to our previous example we added another class, representing the application itself, who takes the concrete integrator class as a parameter to perform the operation. In order to provide the Use Case, this class must own an integrator object that does the job. This object is part of the status of the `Application` class. We say that there is an *aggregation relationship* between an `Application` and an `Integrator`, meaning that the latter is part of the status of the first class. The relationship is represented as line, starting with a diamond on the *container* side. We can, as in this case, specify the *cardinality* of the association, i.e. the number of the objects to be aggregated. In this case any application owns just one integrator. Note that in our design the application knows about integrators, not about its derived classes. Of course, due to the aggregation relationship. we need a method that allows the user to assign the proper integrator to the application. From the diagram one can also tell that the integrator must be passed to the application as a pointer, since the integrator is abstract.

In terms of code one can imagine to have an `Application` object with a method `setIntegrationAlgorithm`, to be use as follows:

```
Application a;
mcIntegrator monteCarloIntegrator;
monteCarloIntegrator.setF(sin);
a.setIntegrationAlgorithm(&monteCarloIntegrator};
a.integrate(a, b);
```

Thanks to polymorphism, we instantiate our preferred concrete integrator, and pass its address to the application. The `Application` object `a` has a pointer `_p` to an `Integrator` besides other attributes. The `Application` can provide an `integrate` method that, in turn, calls the corresponding method of the integrator as `_p->integrate(a, b);`.

9++.2 Design Patterns

What we have just realized is a well known *Design Pattern*, i.e. a known solution to a common problem. Design Patterns are collection of UML diagrams that apply to a large number of problems. There are many compilations of Patterns, that can be found either in the bookshops as textbooks or manuals, as well as in the World Wide Web. Each Pattern is given a name. The Pattern we just derived is known as a *Strategy*.

A Strategy Pattern applies each time a given workflow can be realized with different algorithms. In the Design Patterns language, the abstract integrator is called a *Strategy* and concrete integrators are called *concrete strategies*. The `integrate` method is generically called `AlgorithmInterface`. The application is named a *Context*.

The pattern applies whenever many classes share a common behavior, but differ by the implementation details of some algorithm, or when you need to choose among different algorithms. For example, a program to fit experimental data to a model is usually represented as a Strategy, as well as the integration of differential equations. Consider the problem of studying the results of the integration the equations of motion of an harmonic oscillator with different integration algorithms: it simplifies a lot in terms of a Strategy. The strategy pattern also applies in other completely different fields. For example, suppose you have an application that reads raw data collected with different detectors, with different formats. If each detector provides the same kind of response (i.e. a measurement of a momentum, or energy, or temperature) obtained by a different treatment of the raw data, you can use a Strategy to read abstract data and concrete strategies to transform them into a common format, according to their source.

You should consider using a Strategy whenever you start writing a long sequence of conditional statements in you application.

Chapter 10++

More Design Patterns

In this chapter we continue studying Design Patterns. We show more solutions to known problems. We also introduce a new *tool*, useful to simplify the process of software development: the make utility. This is the right place to introduce this tool, since our problems are solved introducing more and more classes in the game.

10++.1 Planets, again

Consider the problem, shown in Chapter 10 of the textbook, of simulating the gravitational interaction between celestial bodies like the Sun and the planets in our Solar System. In terms of Object Oriented Programming this is a very simple task. What you need, in fact, is an object representing a celestial body, whose status is given by its mass m, its position $\vec{x}$ and its velocity $\vec{v}$. Position and velocity can be represented by a `Vector` class that provides operators for summing vectors, multiplying them by scalars or other vectors, as well as methods to return their length.

A `body` can *move* under the effect of a force F, acting for a period dt. The force F can be represented as a vector, as well. Using the Euler method to integrate the equation of motion we have

```
void body::move(Vector F, double dt) {
  Vector a = F/_mass;
  _v += a*dt;
  _x += v*dt;
}
```

pretty simple, isn't it? Note how concepts expressed in our *natural* language (mathematics) is expressed much like in the same way: the acceleration `a` of a body is just the force `F` divided the body mass. Because of this, its velocity `_v` increases by the acceleration times the interval of time `a*dt`. Similarly, the position changes as $\vec{x}(t+dt) = \vec{x}(t) + \vec{v}dt$.

We can then have an object representing the solar system, whose status consists of a list of bodies. The system evolves for a given amount of time t, in steps of dt. That means that, at each step, for each body i in the list we need to compute the resultant gravitational force acting on i, F_i, due to all other bodies. Such a force is

$$F_i = -\sum_{i \neq j} G \frac{m_i m_j}{r_{ij}^3} \vec{r_{ij}}$$

where $G = 6.673 \times 10^{-11}$ is the Newton constant in SI units, m_k is the mass of body k and $\vec{r}_{ij} = \vec{x}_i - \vec{x}_j$. Evn in this case, the code is relatively simple and is shown in the self–explanatory Listing **??**.

```
void solarSystem::evolve(double maxt, double dt) {
  double t = 0;
  while (t < maxt) {
    list<body>::iterator i = _bodies.begin();
    list<body>::iterator e = _bodies.end();
    while (i != e) {
      Vector F(3);
      F = 0.;
      double mass = i->mass();
      Vector ri = i->x();
      list<body>::iterator o = _bodies.begin();
      while (o != e) {
        Vector r = o->x() - ri;
        double r3 = r.mod2()*sqrt(r.mod2());
        if (r3 > 0) {
          F += r*G*o->mass()*mass/r3;
        }
        o++;
      }
      i->move(F, dt);
      i++;
    }
    t += dt;
  }
  return *this;
}
```

Listing 10++.1 The snippet of code needed to simulate the evolution of many gravitationally interacting bodies.

That's all! You just need to instantiate few objects of class `body` with the proper mass, position and velocity, add them to the solar system and let them evolve for a given time. You can then print the position of each object in the list into a file, to be used to draw the orbits using `gnuplot`.

This is a very simple design, in fact. However, it's unflexible. First of all, if you represent the list of planets belonging to a system as a list of bodies and create those bodies outside the `solarSystem` class, adding them via the proper method, you just put in the list a copy of each body, not the body you created. Besides being a waste of memory, this choice is inappropriate, because at the end of the simulation, bodies in the internal list of the solar system have a status that is different from the one assigned to bodies in your application. The latter have still the initial status, since what is changed is the status of the bodies inside the solar system.

For this reason it is much better to represent the solar system as a list of pointers to bodies. In this way each operation performed inside the solar system, will reflect into bodies created in the application, since the changes in the status happen in the same memory locations.

Moreover, using pointers allows for polymorphism. We can derive objects from bodies that behave like bodies with something more. For example, a spaceship moves like a body when the engines are switched off, but moves differently when we turn on the engines. If we derive a `spaceShip` class from `body`, we can add the spaceship to the solar system, as well, and study how it moves under the effect of both gravitational and internal forces.

Of course, the `move` method for bodies can be realized in different ways, according to the required precision. Then, it is worth describing the body as a Strategy, whose `move` method is the algorithm. There is another solution to this problem: you can define a Strategy by itself and attach the strategy to the body, so that the body becomes a Context, in terms of Pattern language. In your application, then, you choose the appropriate strategy, creates many bodies, to each of which you attach the strategy. Then add bodies (or, better, their pointers) to the solar system and let it evolve for a given amount of time.

10++.2 The Composite Pattern

Consider now the following realization of the problem: we would like to simulate the motion of the Earth around the Sun, as well as the motion of the Moon around the Earth. We can, of course, create three bodies and put them into a solar system. However, since one usually knows the distance and the velocity of a given body relative to another, this approach requires some computation before instantiating the bodies. For example, it is easy to find that the average distance between the Moon and the Earth is $d_M = 384\,399$ km and its average orbital speed is $v_M = 1.022$ km/s. It is also easy to find that the Earth orbits around the Sun at an average distance $d_S = 149\,597\,887.5$ km with an average orbital speed of $v_S = 29.78$ km/s. Let's call $\vec{r}_E = (0, 0, 0)$ the initial coordinates of the Earth and $\vec{r}_M = (d_M, 0, 0)$ the initial coordinates of the Moon. If we want to set the position of the Earth in the solar system in $\vec{R}_E$, with $|\vec{R}_E| = d_S$, we need to translate both the position of Earth and Moon, so that the new positions will be, respectively, $\vec{r}_E + \vec{R}_E$ and $\vec{r}_M + \vec{R}_E$. It is much more simple to describe the operation in the following way:

1. create the Earth and put it at the origin of a system S;
2. create the Moon and place it relatively to the Earth in the same system S;

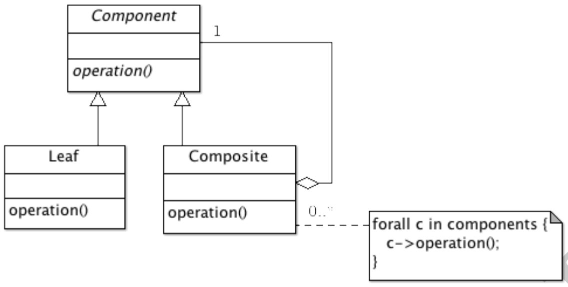

Figure 10++.1 The Composite Pattern.

3. create the Sun and put it at the origin of another system S';
4. add S to S' at a given position.

Of course we can iterate the procedure for every subsystem we may want to include (internal planets, Jupiter with its satellites, Saturn with its satellites and so on).

We can easily see that, in this context, a solar system is a composed object. Its components can be either planets or other (smaller) solar systems. If a solar system is built joining two solar system it becomes a new solar system that can be, in turn, added to another solar system and so on. The solar system is then a *composite*, whose *components* can be other composites or non–composed objects (the planets). We call them *leafs*.

As you can imagine, Composite is one of the most widely used Pattern in Object Oriented Programming, since composite objects appear in fact quite frequent in a variety of problems. The structure of the pattern is illustrated in Figure 10++.1.

It is composed of three classes: a common base abstract class called a *Component*, and two derived classes named a *Leaf* and a *Composite*. An object of the Composite class is composed by Components. Components, in turn, can be either simple, atomic objects (Leafs) or composed objects themselves, i.e., composed of other components. With this pattern each object can be composed of an infinite number of components arranged on an infinite number of different layers.

Given an operation on a component, such an operation returns a value on a Leaf, while returns the result of the aggregation of the results of each component on a composite. For example, suppos we describe a distributed computing system, composed of computing centres, composed in turn of racks, in each of which there are computers, as a Composite. If we ask for the amount of electrical power needed for the whole system we cause the call of its `power` method. Since the system is composed, it loops on centres asking to each of them the `power`, then sums up the results and return the sum. The centres are composed, too. So they return the sum of the power obtained by each rack in the centre.

Again, the rack is composed and returns the sum of the power needed for each of its components. Only at this stage the `power` method of the Leaf (the computer) is called and returns a value.

In our example a self–graviting system can be thought as composed of other self–graviting systems, the leafs being celestial bodies. We then need an abstract class describing a sub system, then a concrete class describing a complex system. In our case the virtual operations are those needed to set the initial status of the objects. Each time we add a system to another system at position $\vec{x}_S$, each of its components must be moved by $\vec{x}_S$.