C++ digest

©Jérôme Novak

January 25, 2008

Contents

1 Introduction

This document is intended to give a few references on the way the C++ language works, for the people attending the school on spectral methods in Meudon in November 2005 and having no knowledge of this programming language. It is very brief (incomplete!) and deals only with most basic syntax and notions (classes, derived classes, virtual methods), with a few examples given in the last section. Interested persons can, of course, consult more complete manuals, like *The C++ Programming Language* by the designer of C++ Bjarne Stroustrup, or refer to one of the many available Web sites, like *e.g* `http://www.cplusplus.com`, which is quite useful when dealing with inputs/outputs (iostream library).

One should not focus too much on the fact that C++ is called an "object-oriented language". It is a programming language with *function* calls and use of *variables*, which can be of different *types*. The notion of class simply gives to the programmer the possibility of defining and introducing his own types; as well as the associated functions to act and interact with other existing types.

As general remarks, it is necessary to declare all classes, functions and variables before they are used or implemented. Except for variables, these declarations are usually put to *header* (or *declaration*) files, which are then included into *source* files that uses or implements them. This implementation is often called *definition* of the function or class (see also Sec. 4). Then, a distinction is made between static and dynamic properties in the program: a *static* feature can be determined or resolved when then program is compiled; whereas a *dynamic* one is completely defined only when the program is executed (for example, it depends on some quantity given by the user at run time).

Finally, all instruction lines must end with a semicolon "`;`" ...

2 Basic syntax

For those who already know about C programming language, it is in principle possible to use any C instruction in C++ too. Although this might be very helpful for people having good skills for inputs / outputs in C, it is however recommended to switch to C++ syntax, as far as memory allocation is concerned, and to forget about `malloc` or `free`...

2.1 Types and variables

A *variable* is the basic brick of the program: it contains information to be processed like *e.g.* numbers. Indeed, some of the simplest *types* of variable, that are pre-defined in C++ are `int`, `float`, `double`, `char`, `bool`, ...:

- `int n ; n = 5 ;` integer number;
- `float x=1.5e10F ;` real floating-point number in single precision; stored on 4 bytes, it describes 8 digits (here $x = 1.5 \times 10^{10}$);
- `double y ; y=2.3e-9 ;` real floating-point number in double precision; stored on 8 bytes, it describes 16 digits (here $y = 2.3 \times 10^{-9}$);
- `char l='w';` a single character (`'\n'`is a newline, `'\t'`a tabulation, ...);
- `bool f=true;` boolean variable that can only be `true` or `false`.

The word "`const`" in front of a type name means that the variable is *constant* and thus its value cannot be modified by the program. Constant variables are initialised when they are declared:

```
const int n = 8 ; // now, writing 'n=2;' is forbidden!
```

Variables can be declared at any point in the code, provided that, of course, they are declared before they are used. The declaration is valid only within the local *block*, *i.e.* within the region limited by braces ("`{      }`"). In the example of Sec. 4.1, the variable `square` is defined only until the first "}", two lines after. It is the local *declaration scope* of variables in C++.

2.2 Pointers and references

A *pointer* on a variable is the address where this variable is stored in the system memory. Pointers can then also be used as variables in the program.

```
int n ;
n = 2 ;
int *p ;
p = &n ;
int x = *p ;
```

In this example, `p` is declared on third line as a variable of type "pointer on an integer" (the type is `int *`); the line after, it is initialised to be the address of the variable `n` (the ampersand meaning "the address of"). Finally, if one wants to use the value stored at the address defined by `p`, a star (`*`) must be put in front of the pointer: on the last line, `x` is initialised to 2. One can therefore see two ways of manipulating variables: through their *values* (like `n` or `x` in this example), or through their addresses (like `p`).

In C++, there is a third way to do so: the *references*. A reference to a variable can be seen as an equivalent to this variable: if one of the couple (variable / reference) is modified, the other is changed too.

```
int n ;
n = 3 ;
int &r = n ;
r++ ; //equivalent to r = r + 1 ;
int x = r - n ;
```

Here, `r` is declared as a reference to an integer (type `int &`) and must be immediately initialised, like `const` variables. In the fourth line, `r` is incremented, and so is `n`, since they are equivalent: every modification to `r` also affects `n`! The results is that, on the last line, `x` is initialised to `4-4=0`.

Constant pointers have a rather different meaning from constant values. `const double *x;` means that the variable pointed by `x` is constant, not `x`! This means that one can write:

```
int n=3;
int p=4;
const int *pn = &n ;
pn = &p ; // OK
*pn = 9 ; //forbidden!
```

One can change the address `pn` (in this case, from the address of `n` to the address of `p`), but one can access the variable pointed by this address in readonly mode. In order to have a constant address, the following syntax must be used:

```
int *const pp = &p ;
```

in that case, an instruction of the type `pp = &n;` is forbidden. Thus, writing `const int *const pq = &n ;` means that both the address `pq` and the variable stored at this address are constant. As for references, only the syntax `const double &x;` has a meaning: `x` can be accessed in readonly mode.

2.3 Functions

In C++, a *function* is a name and a list of arguments; eventually it returns something. All parts of the code (main program, subprograms) are functions and, apart from the main one, they must all be declared before there is a call to them. This declaration is only the statement of (from left to right): the return type (if the functions returns nothing, then it is `void`), the name of the function and, in between parenthesis, a list of the types of its arguments. If there is no argument, then empty parenthesis `()` should be used. An example can be seen in Sec. 4.1, with the function `my_function`, returning a `double` and having as arguments a `double` and an `int`. It is declared before the call in the function `main` (the main program, which always returns an `int`, and which is compulsory to have a code running); it is then defined after this main program. Note that the function `main` does not need to be declared.

Some arguments can be set to default values in the declaration of a function. Consider, for example the declaration of a function that displays an integer in a given base:

```
void display_integer(int num, int base=10) ;
```

so that it is called `display_integer(128, 16)`, to display `128` in hexadecimal base. But with the declaration done above, one can call it also `display_integer(20)` to display `20` in decimal base. The second argument has a *default value*, that need not be specified at the function call. Such arguments (there can be several of them) must always be located at the end of the argument list and, if an argument with a default value has its value specified, all those which are before him must also have their values specified. Please note that default values appear only in the declaration and never in the definition of a function.

As stated at the beginning of this section, a function is specified not only by its name, but also by the list of the types of its arguments. Therefore, it is possible to *overload* a function with another having the same name, but different arguments:

```
int sum(int a, int b) ;
double sum(double x, double y) ;
```

When the function `sum` is called, the choice is made (when compiling) looking at the arguments' types. Note that the return type is not discriminatory and a function `double sum(int, int)` cannot be declared together with the first one of the here-above example.

To end with functions, a few details about arguments and return values are given. A variable given as an argument to a function is not "passed" in the strict sense: the function makes a *copy* of the variable and works on that copy. This means that a function cannot modify a variable, passed as argument! This can be seen on the following example:

```
void swap(int a, int b) { int c=a;
a=b; b=c ; }
int main() {
int n = 2 ; int p = 3 ;
swap(n,p) ; }
```

After the call to `swap`, n=2 and p=3 still! The way of doing here is to pass the variables by their addresses or references on them.

```
void swap(int *a, int *b) { int c= *a;    void swap(int &a, int &b) { int c=a;
*a=*b; *b=c; }                            a=b; b=c ; }
int main() {                           or int main() {
int n = 2 ; int p = 3 ;                   int n = 2 ;  int p = 3 ;
swap(&n,&p) ; }                           swap(n,p) ; }
```

... and everything works fine, since a copy of the address (or reference) still points on (refers to) the same variable. Therefore, if a variable is an output argument of a function, it should not be passed by its value.

2.4 Tests, loops and operators

A simple *test* is written as follows:

```
if (a > 5) {
b = 5 ; //etc ...
}
else { // if needed
b = 3 ;
}
```

The expression following the word `if` must be surrounded by parenthesis and of boolean type: `(n == 2)` equality test, which is different from assignment and has two =’s, `(p != 0)` different from, `(x >= 2.e3)`, ... or a combination of such, using `&&`(logical “and”) or `||`(logical “or”). If the condition is `true`, then the block following it is executed. Eventually, one can put an instruction (followed by a block) `else{...}`. Several other tests are available:

- `do {...} while` (some test)
- `while` (some test) `{...}`
- some test `?` action1 `:` action2 `;`

In this last case (conditional operator), if “some test” is `true` then “action1” is executed, otherwise, it is “action2”.

The operator *switch* is used to choose between different instructions depending on the value of an integer-like variable:

```
switch (n) {
case 0 :
 p = 2 ;
 break ;
case 1 :
 p = 3 ;
 break ;
default:
 p = 0 ;
 l = 3 ;
 break ;
}
```

Finally, a test that is often useful is given by the *assert* command:

```
#include<assert.h>
void my_log(double a) {
assert(a > 0.) ;
... }
```

After the inclusion of its declaration in the file `assert.h`, the use is `assert(` boolean expression `)`. If the boolean expression is true, then the code continues to the next line; otherwise the code is stopped by an `abort()` command. The advantage is that, when compiling the code with the `-DNDEBUG` option, the assert tests are not performed and thus, for an optimised version, it is costless.

The syntax for a *loop* is quite simple and needs three instructions :

1. the initialisation of the loop variable
2. the test (done before loop instructions)
3. the increment of the loop variable (each time the loop is ended)

```
for (int i=0; i<20; i++) {
... //loop instructions
}
```

Here, `i` runs from 0 to 19; note that it is a variable local to the loop *i.e.* it is no longer valid after the loop.

It is also worth mentioning some of the *arithmetic operators*, the last ones in this table can shorten some expressions:

operator	`+, -, *, /`	`%`	`+=` (`a += 5;`)	`-=, *=, /=, %=`	`++` (`a++;`)
meaning	usual arithmetic	module[1]	`a = a + 5;`	similar to `+=`	`a = a + 1;`

2.5 Inputs / Outputs

This section deals with formatted *input*/*output* manipulations. These are done through input- or output-streams and the *iostream* library. The "standard output" (the shell console on which one is typing commands) is called *cout*, and one can send it data thanks to the injection operator "<<":

```
int a = 9 ;
cout << a ;
```

will display '9'. Strings can be displayed directly: `cout<<"Hello world!"<<endl;` where `endl` stands for a new line (and empties the buffer). More about strings is given in Sec. 2.6. All standard types (see Sec. 2.1) can be outputted in this way, without specification of the format to be used. In order to change the format (precision, fixed / scientific, *etc* ...), *manipulators* are employed (see `http://www.fredosaurus.com/notes-cpp/io/omanipulators.html`). Standard input (from the keyboard, in the console) is accessed through *cin*:

[1]module being the operation that gives the remainder of a division of two integer values

```
cout << "Enter a number:" << endl ;
int n ;
cin >> n ;
```

Any function using `cin`, `cout` or `>>`-like operators must have declared the iostream library and must use the standard namespace (see Sec. 2.7) by adding the following two lines before its definition:

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

Accessing to files is done in a very similar way:

```
double x = 1.72e3 ;
ofstream my_file("output.dat") ;
my_file << "The value of x is: " << x << endl ;
```

this opens a file called "output.dat" (created, if it does not exist, erased if it does) and writes things into it. `my_file` is an object of type *ofstream* (output file stream), linked with the file opened in write-mode. Similarly, to read data from a formatted (existing) file, one should do as for reading from the standard input:

```
ifstream a_file("input.dat") ;
int a ;
a_file >> a ;
```

In this case, `a_file` is of type *ifstream* (input file stream). Before using any of these types, one should declare, in addition to the iostream library and the standard namespace, the fstream one with a "`#include<fstream>`".

2.6 Memory allocation

There are two ways of defining an *array* in C++: static and dynamic allocation. The static way is *e.g.* `double tab[37] ;` (note that the indices of `tab` range from 0 to 36), or with constant integer variable for the dimension `const int nsize = 100; int tbl[nsize];`. Here the size of each array is known at compilation time. On the contrary, when this size cannot be known, one must use *dynamic memory allocation*:

```
int n ;
cout << "Enter size" << endl ;
cin >> n ;
double *tab = new double[n] ;
```

In this case, `tab` is an array, which elements can be accessed as before: from `tab[0]` to `tab[n-1]`. The syntax used here shows that an array can be seen as a pointer on its first element, so there is an implicit compatibility between arrays and pointers. The allocation of the memory is done at runtime thanks to the instruction *new*, but this memory must then be given back to the system, using the instruction *delete*, when the array is no longer used: `delete [] tab;`

The simplest implementation of *strings* is done through arrays of `char`s, with the use of double quotes, contrary to single `char`s:

```
char* var = "Hello!";
```

another example is given is Sec. 4.1. Note that such strings end with the character `'\0'`, so here `var` has seven elements.

A full program implementing many features described here is shown is Sec. 4.1.

2.7 Static variables and namespaces

A *static variable* keeps its value from one function call to the next:

```
void f() {
static int n_call = 0 ;
if (n_call == 0) { ... } //first call operations
n_call++ ; ...
```

In this example, `n_call` is initialised to 0 when the function is called for the first time, but it then keeps its value (it is no longer initialised!) at next function calls. So, in this case, the value of `n_call` is the actual number of calls to `f()`.

However, this kind of syntax can be replaced in C++ with the use of a *namespace*. This is a declaration region that can be used in several functions, without interfering with local variables:

```
namespace my_name {
int i, a ;
void g() ;
}
```

Here is declared a namespace called `my_name` with `i, a` and the function `g` as members. After including the file containing this namespace, one can use some of its variables `my_name::i = 0;`, or the whole namespace:

```
using namespace my_name ;
g() ;
```

In such case there is no need to specify `my_name::`, the call to `g()` means `my_name::g()`. In Lorene, namespaces are used to carry unit definitions (numerical constants). Finally, in order to replace static variables, one should use an *anonymous namespace*:

```
namespace {
int n_call = 0 ;
}
void f() {
if (n_call ... }
```

in particular, there is no instruction `using`.

3 Classes

A *class* is a collection of data and functions. It generalises the notion of type by giving the possibility to the programmer to define new types of his own. In particular, one can overload (see Sec. 2.3) standard operators (*e.g.* arithmetic operators, or output) with these new types, as it is shown in complete examples of Sec. 4.2 and 4.3. Once a class is declared and defined, one can declare variables of that new type, use pointers on it or references to it, as if it were a standard type (double, int, ...). Classes has already been used in this document: output files were declared as `ofstream` in Sec. 2.5, which is a C++ class...

3.1 Members

To declare a class, one must specify its *members*: data (variables of other types, including eventually other classes) and functions, sometimes called *methods*. The syntax is:

```
class My_class {
int n ;
double x ;
double f(int) ;
} ;
```

This declares a class called `My_class`, with two data members, called `x` and `n`, and one method `f(int)`. Actually, the full name of the method is (to be used when defining it outside the class declaration):

```
double My_class::f(int p) {
x = 2.3*(p+n) ;
double res = 3./x ;
...}
```

Methods of a class can use the data without need to re-declare them: here `n` or `x` are known to be members of `My_class`.

In a function using this class, these members are used the following way (one must include the declaration of the class before using it)[1]:

```
My_class w ;
int q = w.n + 2 ;
cout << w.f(q) ;
```

The variable `w` is an object of type `My_class`, and has its own members that can be accessed through the operator "`.`". When considering pointers, the operator is "`->`":

```
My_class *v ;
double y = v->x + 0.2 ;
cout << v->f(3) ;
```

Usual arithmetic operators can be overloaded to work with this new class:
e.g. `My_class operator+(My_class, My_class);` for the declaration, and, in some other function:

```
My_class z1 ;
My_class z2 ;
My_class t = z1 + z2 ;
```

3.2 Restriction of access and `const` issues

A central notion when manipulating classes is that some of its members are not accessible (*i.e.* usable) by "normal" functions, (functions which are not member of the same class. This is called *restriction of access* and is achieved through the keywords `public`, `private` and `protected`:

[1]the first line assumes that there is a constructor (see Sec. 3.3) without parameters

```
class My_class {
private:
int n ;
double x ;
public:
double f(int) ;
private:
void g() ;
} ;
```

In this case, data and the method `My_class::g()` are private, meaning that only functions member of `My_class` can use them, therefore the last two lines of

```
My_class v ;
cout << v.n ;
v.g() ;
```

are forbidden in all other functions. On the contrary, a call to `v.f(2)` is allowed. By default, all members of a class are private, so it is highly recommended to use the keywords when declaring a class. Such a keyword is valid until the use of another one; and `protected` has the same effect as `private` in a class, the difference appearing only for derived classes (Sec. 3.4).

Exceptions can be made declaring some functions or other classes to be *friend*, within the declaration of the class:

```
class My_class {
private:
int n ; ...
friend double ext_f() ;
} ;
```

Then, inside the definition of the non-class member `double ext_f()`, one can access private and protected members of `My_class`. A class can be declared friend the same way: adding `friend class Other_class ;` in the declaration of `My_class` and, inside all methods of `Other_class`, it is then possible to access to private/protected members of `My_class`.

The notion of *constant* object (see Sec. 2.1) means for a class that all data of this object are constant. Exceptions are possible with the keyword `mutable`: a *mutable* data member is a member that can be modified, although the object it belongs to is seen as constant. The idea can be that mutable members are somehow "secondary" members, that can be deduced from other "primary" data. So, as long as these primary data are not modified, the object is supposed constant. The syntax is, within a class declaration:

```
class My_class {
public:
int n; ...
mutable double sec ;
} ;
```

So, in a function using `My_class`, one has then:

```
const My_class w = ... ;
w.n = 2 ;                // forbidden!
w.sec = 0.7 ;            // OK
```

A member function that keeps the object it is called on constant is said constant too, the keyword `const` can be added at the end of its declaration and definition:

```
class My_class {
...
double f(int) const ;
void g() ;
... };
```

Here, `double My_class::f(int)` is constant, whereas `void My_class::g()` is not. As before, in a function using this class:

```
const My_class w = ... ;
cout << w.f(2) ;           //OK
w.g() ;                    //forbidden!
```

3.3 Constructors, destructor and assignment operator

When designing a new class, special care must be devoted to four particular member functions:

- a *standard constructor* – this function is called to create an object of this class. It has the same name as the class, and no return type (*e.g.* `My_class::My_class()`. It can take arguments or not and its task can be (not compulsory at all) to initialise data members or to allocate memory.

- a *copy constructor* – creates an object from an existing one. It is also a constructor as the standard one but it must take exactly one argument of the type `const My_class &`, meaning that it needs a reference on an object of that class that will not be modified to build the new object (readonly!).

- a *destructor* – destroys the object when it is no longer valid, *i.e.* at the end of its declaration scope. Its task is mainly to check if there is some dynamically allocated memory to be given back to the system. It has the same name as the class, but with a tilde (˜) in front; it takes no argument and has no return type.

- an *assignment operator* – used to assign an existing object to another one, with an instruction like `a=b`. Its name is `My_class::operator=` and, as the copy constructor, it takes one argument of the type `const My_class &`.

These four methods are compulsory to have a usable class. Note that there can be several constructors as long as there is a copy constructor and a standard one. A “fifth” function is very useful: an overload of the `<<` operator to display objects of the class. This function is necessarily an external one (*i.e.* it is not a member of the class) and can also be used to output to files instead of `cout`. A full example implementing all these functions, together with a main function is shown in Sec. 4.2, with the class `rational`, representing rational numbers. In particular, in the definition of this class (Sec. 4.2.2), both constructors use the *initialisation list* that directly initialises class data from the constructor's arguments

```
rational::rational(int a, int b) : num(a), denom(b) { ...}
```

Here, data `num` and `denom` are directly copied from the variables `a` and `b`; *i.e.* it is equivalent to writing

```
 rational::rational(int a, int b) {
num = a ;
denom = b ; ... }
```

Still, the initialisation list is more readable and will be used for derived classes (see next section).

Another example is given by the class `My_array` in Sec. 4.3, with dynamic memory allocation (*i.e.* a non-trivial destructor). Only the main structure is given, but the class can be compiled.

3.4 Derived classes

An existing class can be completed into a new class, which then has more members. This is the *inheritance* mechanism that allows the (new) *derived class* to get the properties of the (existing) *base class*, and to add new ones. The declaration of the new class is done as follows (once the class `My_class` has been declared):

```
class New_class : public My_class {
int p ;
int h() ; }
```

In this example, the class `New_class` is derived from `My_class`, to which it adds two new members.

Actually, the derived class does not inherit all members of the base class. First, the private members are not accessible to the derived class methods (whereas protected ones are! This is the difference between private and protected access); then, none of the assignment operators (which name are `operator=`), constructors or destructor are inherited. Therefore, one must re-declare and re-define these members, but with the help of their equivalents in the base class. In particular, for the constructors of the derived class, one must first call the constructor for the base class, through the initialisation list. Then, the new members of the derived class are initialised in this list and, finally, other actions are performed in the body of the constructor. The destructor for the new class works in the opposite way: first the new members must have their memory given back to the system (if any) and, at the end, there is automatically a call to the destructor of the base class. A simple example is given in Sec. 4.3.4, with the new class `Square_matrix` being a derived class from `My_array`, but with no new data member.

A very important point is that there is an *implicit compatibility* between the derived class and the base class. This is valid only for pointers and references to the derived class, which can be used instead of pointers or references to the base class.

```
My_class *w ;
New_class *q ;
w = q ;
```

is allowed, whereas `q = w;` is forbidden. After the third line above, the static type of `w` is `My_class *` (obtained from the declaration), whereas the dynamic type is `New_class *`, since it is pointing on an object of this derived class. This dynamic type cannot, in general, be determined at compilation time (imagine there is a test depending on some reading from `cin`, to decide whether `w = q` is invoked or not). If one wants to know, in this example, the type of `w`, a possibility is to use the instruction *dynamic_cast*:

```
New_class *z = dynamic_cast<New_class *>(w) ;
```

In that case, if the dynamic type of `w` is `New_class *`, then `z != 0x0` (`z` is not the null pointer). Actually, this is the case if `w` is compatible with the type; *i.e.* `z` is not null also if `w` is a pointer on a derived class of `New_class`.

The equivalent for references can be seen in the assignment operator of `Square_matrix` (Sec. 4.3.4), where there is a call to the assignment operator of the base class `My_array`, but with a reference to an object of the derived class as argument, instead of a reference to an object of the base class (see the declaration of `My_array::operator=`).

3.5 Virtual methods

In the examples cited above, a problem can arise if the derived class re-declares a method of the base class:

```
class My_class {
double f(int) ; ...} ;
class New_class : public My_class {
double f(int) ; //different from that of My_class
...} ;
My_class *w ;
New_class *q ;
int v ; cin >> v ;
if (v==0) {
w = q ; }
cout << w->f(2) ;
```

Which method `f(int)` is called ? This is impossible to determine at compilation time, but is not an academical problem since, in each derived class, this is exactly the case for destructors. In the above example, it might happen that not all the memory allocated to `w` is freed. Therefore, there is a mechanism in C++ called *polymorphism* that makes the link with the right function at execution time (dynamically). It is obtained by the use of the keyword *virtual*, for the declaration, in front of such “ambiguous” methods:

```
class My_class {
virtual double f(int) ; ...} ;
class New_class : public My_class {
virtual double f(int) ;
...} ;
```

Now, everything works fine and the call is done to the right function. The only requirement is that the list of arguments must be the same for all the virtual functions having the same name. Note that, every time inheritance is used, one must declare all destructors of base / derived classes as `virtual`. Another example is given in Sec. 4.3.3 with the method `display(ostream&)`: the standard display is achieved through the call to `operator<<` which, thanks to implicit compatibility, can also be called with (reference to) `Square_matrix` objects. This function then calls to the virtual method `display(ostream&)`, which gives different output, depending on the type of `tab_in`.

3.6 Abstract classes

With the possibility of deriving classes, it is sometimes interesting to have some classes that are not actually usable, but that can be used as templates for the design of other classes. These classes therefore possess one or several methods that are too general to be defined (implemented): in LORENE, this is the case *e.g.* for a general equation of state. Such kind of functions are then declared as *pure virtual method*. The declaration is then ended with a "`=0 ;`" and no definition is given:

```
 class Eos {
virtual double p_from_rho(double) = 0 ; ... };
```

Still, a derived class, which is usually more specific, can implement that method, using polymorphism:

```
 class Eos_polytrope: public Eos {
virtual double p_from_rho(double)  ; ... };
```

In the example of `Eos`, one cannot declare an object of this type, since the class is incomplete, only a derived class which implements the pure virtual methods can be used. Nevertheless, one can declare a pointer or a reference to an `Eos`:

```
Eos eo ; //forbidden
Eos_polytrope ep ; //allowed, it implements p_from_rho
Eos *p_eos = &ep ; // OK, not instance +  implicit compatibility
Eos &r_eos = ep  ; // OK, not instance +  implicit compatibility
```

`Eos` is called an *abstract class*, for one cannot declare any *instance* (no direct objects, only pointers or references to) of this class. More generally, since an abstract class is a class which cannot be instantiated, these are classes that:

- have a pure virtual method;
- derive from a class with a pure virtual method that they do not define;
- have only private or protected constructors.

4 Examples

4.1 A first program

This program does not do any interesting job, it is just an illustration of the basic syntax in C++.

```
// C++ headers
#include <iostream> //<> are for system headers, "" for user-defined ones
#include <fstream>
// C headers
#include <math.h> // in principle, C headers contain a .h, whereas C++ do not
using namespace std ;  // to get input / output objects (cin, cout, ...)
double my_function(double , int) ; // local prototype (declaration only)

int main(){  // In every executable there must be a main function returning an integer
  const int nmax = 200 ;
  double stat_array[nmax] ; //static allocation of memory
  char dim[] = "size" ;
  cout << "Please enter a "<< dim << " for an array between 1 and 200" << endl ;
  int dyn_size ;
  cin>>dyn_size ;
  if ((dyn_size<1)||(dyn_size>200)) {
    cout << "the " << dim <<" must be between 1 and 200!" << endl ;
    cout << "try again: " ;
    while ((dyn_size<1)||(dyn_size>200)) cin>>dyn_size ;
  }
  double *dyn_array = new double[dyn_size] ; //dynamic memory allocation
  for (int i=0; i<nmax; i++) {
    int square = i*i ;
    stat_array[i] = square ;
  }
  double cube ;
  for (int i=0; i<dyn_size; i++) {
    cube = pow(double(i),3) ; //Conversion of an integer to a double
    dyn_array[i] = cube + my_function(stat_array[i], dyn_size) ;
  }
  cout << "The value of the variable dyn_array is: " << dyn_array << endl ;
  cout << "its first element is: "<< *dyn_array
       << " or, alternatively: " << dyn_array[0] << endl ;
  cout << "Saving dyn_array to the file exa1.dat..." << endl ;
  ofstream output_file("exa1.dat") ;
  for (int i=0; i<dyn_size; i++) {
      output_file << i << '\t' << dyn_array[i] << '\n' ; }
  output_file << endl ;
  delete[] dyn_array ; // It is necessary to release the allocated memory...
  return EXIT_SUCCESS ; // If the program came up to here, everything went fine
}
// definition of "my_fonction"
double my_function(double x, int n) {
  double resu = log(x+double(n)) ;
  return resu ;
}
```

4.2 A class of rational numbers

To compile it, just type (*e.g.* with the GNU C++ compiler):

```
   g++ -o ratio ratio.C rational.C gcd.C
```

4.2.1 Declaration file rational.h

```
#ifndef __RATIONAL_H_        // to avoid multiple declarations
#define __RATIONAL_H_
#include <iostream>          // ostream class is used
using namespace std ;
class rational {             // beginning of the declaration of class rational
    // Data:
    // -----
```

```
 private:
    int num ;                  //  numerator
    int denom ;                //  denominator

    //Required member functions
    //------------------------
 public:
    rational(int a, int b = 1) ; // Standard constructor to create a/b
    rational(const rational& ) ; // Copy constructor

    ~rational() ; // Destructor

    void operator=(const rational&) ; //Assignment from another rational

    // Data access
    // -----------
    int get_num() const {return num ; };       //inline definition
    int get_denom() const {return denom ; } ;  //inline definition

    //Display: declaration of "friendness" only
    friend ostream& operator<<(ostream& , const rational& ) ;
}; // end of the declaration of class rational

//True declaration of the function, not member of the class
ostream& operator<<(ostream& , const rational& ) ;
// External arithmetic operators to calculate expressions such as 'p + q*r'
rational operator+(const rational&, const rational&) ; // rational + rational
rational operator-(const rational&, const rational&) ; // rational - rational
rational operator*(const rational&, const rational&) ; // rational * rational
rational operator/(const rational&, const rational&) ; // rational / rational
#endif
```

4.2.2 Definition file `rational.C`

```
// Include files
#include <assert.h>
#include "rational.h"

int gcd(int, int) ; //local prototype of an external function (greatest common divisor)

//--------------//
// Constructors //
//--------------//
// Standard
rational::rational(int a, int b):num(a), denom(b) {
  assert(b!=0) ;

  if (num == 0) denom = 1 ;
  else {
    int c = gcd(a, b) ;
    num /= c ;
    denom /= c ;
  }
}

// Copy
rational::rational(const rational & rat_in): num(rat_in.num), denom(rat_in.denom)
{
  assert(rat_in.denom != 0) ;
  assert(gcd(num, denom) == 1) ;

}
//------------//
// Destructor //
//------------//
rational::~rational() { }

//------------//
// Assignment //
//------------//
// From rational
void rational::operator=(const rational & rat_in) {
  assert(rat_in.denom != 0) ;
  num = rat_in.num ;
  denom = rat_in.denom ;
  assert(gcd(num, denom) == 1) ;
}

//---------//
// Display //
//---------//
// Operator <<
ostream& operator<<(ostream& o, const rational & rat_in) {

    if (rat_in.denom == 1) o << rat_in.num ; //as a friend it can access private data
  else
    o << rat_in.num << "/" << rat_in.denom ;
  return o ;
}

                       //-----------//
                       // Addition  //
                       //-----------//

// rational + rational, not friend, must use access functions
rational operator+(const rational& t1, const rational& t2) {
  rational resu(t1.get_num()*t2.get_denom() + t2.get_num()*t1.get_denom(),
t1.get_denom()*t2.get_denom()) ;
  return resu ;
}
```

4.2.3 GCD function

```
int gcd(int a, int b) {
  if (a<b) {
    int c = a ;
    a = b ;
    b = c ;
  }
  int reste = a%b ;
  while (reste != 0) {
    a = b ;
    b = reste ;
    reste = a%b ;
  }
  return b ;
}
```

4.2.4 Main program `ratio.C`

```
//Declarations of the class rational
#include "rational.h"

int main(){
    rational p(420,315) ;   // 420/315, simplified by the constructor
    rational q(5) ;         // 5/1
    cout<< p + q<< endl ;   // call to operator+ and operator<<
  return EXIT_SUCCESS ;
}
```

4.3 Classes `My_array` and `Square_matrix`

Although the classes `My_array` and `Square_matrix` can be compiled, they have incomplete features to be used on some real example. Only declaration and definition are given for better clarity.

4.3.1 Declaration file my_array.h

```
#ifndef __MY_ARRAY_H_
#define __MY_ARRAY_H_

#include<iostream>
#include<assert.h>
```

```
using namespace std ;

class My_array {
    // Data :
    // ------
 protected:
    int size1 ;        //size in first dimension ...
    int size2 ;
    int size3 ;
    double* tableau ;  // the actual array

    // Constructors - Destructor
    // ---------------------------
 public:
  explicit My_array(int dim1, int dim2=1, int dim3=1) ; //standard constructor
  My_array(const My_array&) ; //copy constructor

  virtual ~My_array() ; //destructor

    // Assignments
    // -----------
  void operator=(const My_array&) ; //assignment from another My_array

    // Data access (inline)
    // --------------------
  int get_size1() const {return size1 ; };
  int get_size2() const {return size2 ; };
  int get_size3() const {return size3 ; };

  double operator()(int i, int j=0, int k=0) const { //read-only access (const)
      assert ((i>=0) && (i<size1)) ;    // tests: are we beyond array bounds?
      assert ((j>=0) && (j<size2)) ;
      assert ((k>=0) && (k<size3)) ;
    return tableau[(i*size2 + j)*size3 + k] ;
  };

  double& set(int i, int j=0, int k=0) {  //read-write access (thanks to the reference!)
    assert ((i>=0) && (i<size1)) ;
    assert ((j>=0) && (j<size2)) ;
    assert ((k>=0) && (k<size3)) ;
    return tableau[(i*size2 + j)*size3 + k] ;
  };

 protected:
  virtual void display(ostream& ) const ; //to use polymorphism

  // External function to be called for the display
  friend ostream& operator<<(ostream&, const My_array& ) ;
};

ostream& operator<<(ostream&, const My_array& ) ;
#endif
```

4.3.2 Definition file my_array.C

```
#include<fstream>  // to manipulate file streams ...
#include<iomanip>  // ...  and output format.
#include "my_array.h"

My_array::My_array(int dim1, int dim2, int dim3) : size1(dim1), size2(dim2), size3(dim3),
 tableau(0x0) {
  assert((dim1>0)&&(dim2>0)&&(dim3>0)) ;
  tableau = new double[dim1*dim2*dim3] ;
}

My_array::My_array(const My_array& tab_in) : size1(tab_in.size1), size2(tab_in.size2),
      size3(tab_in.size3), tableau(0x0) {

    assert((size1>0)&&(size2>0)&&(size3>0)) ;
    int t_tot = size1*size2*size3 ;
    tableau = new double[t_tot] ;
    assert(tab_in.tableau != 0x0) ;
    for (int i=0; i<t_tot; i++)
      tableau[i] = tab_in.tableau[i] ;
}

My_array::~My_array() {
  if (tableau != 0x0) delete [] tableau ;
}

void My_array::operator=(const My_array& tab_in) {
  assert(size1 == tab_in.size1) ;
  assert(size2 == tab_in.size2) ;
  assert(size3 == tab_in.size3) ;
  assert(tab_in.tableau != 0x0) ;
  assert(tableau != 0x0) ;

  int t_tot = size1*size2*size3 ;
  for (int i=0; i<t_tot; i++)
    tableau[i] = tab_in.tableau[i] ;
}

void My_array::display(ostream& ost) const {
  assert(tableau != 0x0) ;

  ost << "My_array: \n";
  ost << size1 << "x" << size2 << "x" << size3 << " elements" << endl ;
  ost << setprecision(5) ;

  for (int i=0; i<size1; i++) {
      ost << "i=" << i << '\n' ;
      for (int j=0; j<size2; j++) {
  for (int k=0; k<size3; k++) {
      ost << tableau[(i*size2+j)*size3 + k] << '\t' ;
  }
  ost << endl ;
      }
      ost << endl ;
  }
  ost << endl ;
  return ;
}

ostream& operator<<(ostream& ost, const My_array& tab_in ) {
  assert(tab_in.tableau != 0x0) ;
  tab_in.display(ost) ;
  return ost ;
}
```

4.3.3 Declaration file matrix.h

```
#ifndef __SQUARE_MATRIX_H_
#define __SQUARE_MATRIX_H_
#include "my_array.h"

class Square_matrix: public My_array { //inherits from My_array

    // Constructors - Destructor
    // --------------------------
 public:
    explicit Square_matrix(int ) ;  //standard constructor
    Square_matrix (const Square_matrix& ) ; //copy constructor

    virtual ~Square_matrix() ; //destructor (virtual, as needed)

    // Assignment
    // -----------
    void operator=(const Square_matrix&) ; //assignment from another Square_matrix

 protected:
    virtual void affiche(ostream& ) const ; //Display (virtual)
};
```

```
#endif
```

4.3.4 Definition file matrix.C

```
#include<iomanip>           //to have the manipulator setprecision()
#include "matrix.h"

// Default constructor
//--------------------
Square_matrix::Square_matrix(int dim1) : My_array(dim1, dim1) {
  assert(dim1>0) ;
}

// Copy constructor
//-----------------
Square_matrix::Square_matrix(const Square_matrix& tab_in) : My_array(tab_in) {}

// Destructor (does nothing, since there is an implicit  call to ~My_array() )
//-----------
Square_matrix::~Square_matrix() {}

// Assignment operator
//--------------------
void Square_matrix::operator=(const Square_matrix& tab_in) {
  My_array::operator=(tab_in) ;
}

// Display
//--------
void Square_matrix::affiche(ostream& ost) const {
  assert(tableau != 0x0) ;

  ost << "Square_matrix  " << size1 << "x" << size2 << endl ;
  ost << setprecision(5) ;
  for (int i=0; i<size1; i++) {
    for (int j=0; j<size2; j++) {
      ost << tableau[i*size2 + j] << '\t' ;
    }
    ost << endl ;
  }
  ost << endl ;
}
```

Index