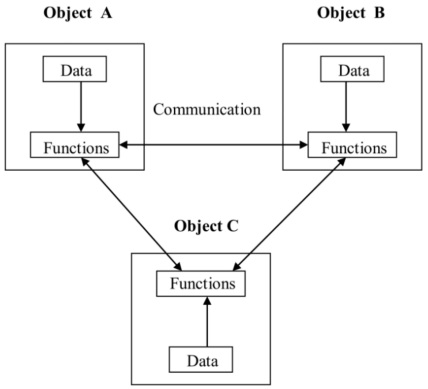
Object-oriented and software design
Introduction to C++

Laurence Pierre

I. CLASSES AND ENCAPSULATION IN C++

C++ has been designed and implemented by Bjarne Stroustrup and it has been *standardized in 1998*, as *ISO/IEC 14882-1998*. Then the standard has been updated several times (https://isocpp.org/std/the-standard). Here we study the **fundamental principles**, and not the various technical developments of the different standards.

C++ is a representative language of **object oriented programming**. Here is a pictorial view of this programming paradigm:



https://dev.to/sagary2j/high-level-object-oriented-programmingoop-concepts-f0b

→ **Objects** (instances of "classes") contain **data** that are encapsulated (hidden), and a program consists of a set of **objects** that communicate with each other by "sending messages" i.e., calling **functions** ("methods") of other objects.

Example : Simple example of an application that stores a set of integers, and allows to search for the presence of an element in this set.

Imperative paradigm (C):

```
void sort_array(int* t, int size) {
  int i, j, current;
  for (i = 1; i < size; i++) {
    current = t[i];
    for (j = i; j > 0 && t[j - 1] > current; j--) {
      t[j] = t[j-1];
    }
    t[j] = current;
  }
}

int binary_search(int *arr, int size, int target) {
  int low = 0, high = size - 1, mid;
  while (low <= high) {
    mid = (low + high) / 2;
    if (arr[mid] == target) {
      return mid;
    } else if (arr[mid] > target) {
      high = mid - 1;
    } else {
      low = mid + 1;
    }
  }
  return -1;
}

void fill_array(int* t, int size) {
   ...
}

int main(){
  int T[10];   // array of 10 elements
  int x;
  // and the application is in charge of this array:
  fill_array(T,10);
  sort_array(T,10);
  ...
  printf("Which element do you want to search?  ");
  scanf("%d", &x);
  printf("result: %d\n", binary_search(T,10,x));
  return 0;
}
```

Object-oriented paradigm (C++):

```
class my_collection {
private:
  int *T, size;   // private data of the class
  void sort_array() {
    // sorts the attribute T
    ...
  }
  int binary_search(int target) {
    // searches for target in T
    ...
  }
```

```
public:
  my_collection(int s){
    // allocates memory for T of size s
    ...
  }
  void fill_collection() {
    // fills the attribute T
    ...
  }
  int search(int target) {    // naive version ☺
    sort_array();
    return binary_search(target);
  }
};

int main() {
  my_collection c(10);
  int x;
  // the object is in charge of its data:
  // (the application doesn't even know that the attribute
  //  is an array, nor how it is handled!)
  c.fill_collection();
  printf("Which element do you want to search?  ");
  scanf("%d", &x);
  printf("result: %d\n", c.search(x));
  return 0;
}
```

I.1 Classes and instances

A ***class*** defines a family of objects that have the **same structure and the same behaviour**. It contains ***data*** (*attributes*) and ***methods*** (member *functions*). A class definition can be considered as a model that is used to create ***instances*** of that type.

I.1.1 Class definition

✓ The syntax to define a C++ class is as follows:

```
class identifier {
      private :
         < data and methods >    // private access
      protected :
         < data and methods >    // access allowed for subclasses
      public :
         < data and methods >    // public access
};
```

C++ gives the possibility to define ***3 sections*** *in a class*:

- **private** section: these data and functions are not accessible from outside the class, they are accessible only from the methods of the class
- **protected** section: only subclasses can access these data and functions
- **public** section : no restriction. In general, only functions are public.

Example : Very simple class that defines a counter

```
class counter {
  private :
     unsigned int value;    // attribute
  public :
     // public methods:
     counter() {    // constructor
        value = 0;  // value initialized to 0
     }
     void increment() {  // increment the counter
        value++;
     }
     void decrement() {  // decrement the counter
        value--;
     }
     unsigned int get_value() {  // "getter"
        return value;
     }
};
```

From the outside world, the **private** variable `value` is only accessible through calls to the methods ("message sending") `increment`, `decrement`, and `get_value`.

The first method above has the same name as the class, this is the ***constructor***: it is *automatically executed when an instance of the class is created*. In the example, it initializes the counter value to 0.

I.1.2 Instances

The **creation** of a (static - non "dynamic" i.e., pointer) ***instance*** is *similar to the declaration of any variable in C*. Additionally, the constructor has the possibility to receive parameters (see later). Here is an example in which two instances `c1` and `c2` of class `counter` are created. This is a special simplified case here: the attribute is initialized using the constructor *without parameters*.

Note that communication with class instances is made by ***message passing***, which uses the **.** operator.

Example :

```
#include <iostream>

using namespace std;

int main () {
counter c1, c2;       // constructor automatically called
for (int i=1; i<=15; i++) {
   c1.increment();
   cout << "Value of c1: " << c1.get_value() << endl;
   c2.increment();
   ...
}
for (int i=1; i<=5; i++)
   c2.decrement();
...
return 0;
}
```

Add-on. A few words about inputs/outputs

✓ It is possible to use functions `printf` and `scanf`, but in general interactive I/O are performed with the ***operators << and >>*** which do not require to specify a format. In that case, the header file `iostream.h` is included.

The **<<** operator inserts data into the stream that precedes it. In particular, `cout` represents the standard output, and `cerr` represents the standard error output. The type of both of them is `ostream`.

The **>>** operator extracts data from the stream that precedes it. In particular `cin` represents the standard input. The extraction operation uses the type of the variable after the >> operator to determine how it interprets the characters that are read (if it is an integer then a series of digits is expected, if it is an string then a sequence of characters is expected,…).

Examples :

```
int i;
cout << "Give the value of i\n";
cin >> i;
cout << "Here is the value of i: " << i << endl;
```

Both operators can be used with the predefined types short, int, long, float, double, bool, char and char *.

✓ *Manipulators* can be used to change formatting parameters on streams and to insert or extract certain special characters. For instance `endl` inserts a newline ('\n') and flushes the buffer. The use of decimal, hexadecimal or octal bases is specified using the manipulators `dec`, `oct`, and `hex`.

Example :

```
int i = 24, j = 91;
cout << "Hexadecimal value of i: " << hex << i << endl;
cout << "and of j: " << j << endl;
cout << "Oops !... decimal value of j: "
     << dec << j << endl;
```

which results in:

```
Hexadecimal value of i: 18
and of j: 5b
Oops !... decimal value of j: 91
```

✓ Finally let us mention the function `eof()` that enables to check the end of file. However, it can be sufficient to *check the value of the extraction operation*: the value of this expression is *true as long as the end of file is not reached*.

Thus it is possible to have a loop as follows:

```
char c;
while (cin >> c)
   cout << "extracted character: " << c << endl;
```

To consider every kind of characters (including white space), it is better to do as follows:

```
while (cin.get(c))
   cout << "extracted character: " << c << endl;
```

and to read line per line:

```
char input_line[128];
while (!cin.eof()){
  cin.getline(input_line,128);
  cout << "extracted line: " << input_line << endl;
}
```

I.1.3 Modular declaration and definition

To make it simple in the example of section I.1.1, the *declaration* and the *definition* of the class are made as a single block, in the same file. But it is not usual. A usual architecture in C++ is *modular*, a header file and an implementation file are associated with each class (or family of classes).

The **header** file contains the **declaration** of the class, it may also contain declarations of *independent* functions that are related to this class. Note that each function must be correctly documented (use Doxygen, http://www.doxygen.nl).

The **implementation** file contains the **definitions** of the functions.

/!\ Since the definitions of the functions will be given ouside the class, the *scope resolution operator* **::** must be used.

Example :

```
// File "count.h" (header file)
     class counter {
       private :
          unsigned int value;
       public :
          counter() ;
          void increment() ;
          void decrement() ;
          unsigned int get_value() ;
     };

// File "count.cpp" (implementation file)
     #include "count.h"

     counter::counter() {
        value = 0;
     }
     void counter::increment() {
        value++;
     }
     void counter::decrement() {
        value--;
     }
     unsigned int counter::get_value() {
        return value;
     }
```


I.2 Constructors

I.2.1 Constructors with parameters

Constructors can have **parameters**, to enable different initializations of the attributes for different class instances.

Example : let us come back to the counter example

```
class counter {
  private :
     unsigned int value;
  public :
     counter(unsigned int v) {   // const. with parameter
        value = v;
     }
     void increment() {
        value++;
     }
     void decrement() {
        value--;
     }
     unsigned int get_value() {
        return value;
     }
 };
```

I.2.2 Creation of instances

When the constructor has parameters, the ***creation of an instance*** **must** give **actual parameters** to the constructor. For a static (i.e., non dynamic) instance, these parameters are given *between parentheses after the identifier of the variable*, for **example**:

```
int main () {
   counter c1(10);
   for (int i=1; i<=15; i++){
      c1.increment();
      cout << "c1 contains: " << c1.get_value() << endl;
   }
   return 0;
}
```

Note. The parameters of a constructor, like for any C++ function, can have ***default values***. It is also possible to define ***several constructors*** for a same class.

This can be particularly useful when *arrays* of class instances are used: for their

declaration, it is mandatory to *call the constructor without arguments*.

Example :

```
class point {
  private :
    int x1, x2;
  public :
    point(int x, int y) {
       x1 = x;  x2 = y;
    }
    ...
};
```

With such a constructor, it is possible to write, for example `point p(3,4);`

but neither `point p2;`

nor `point tabp[20];  // array declaration`

For example, in the first case, we get the following error message when compiling

```
test_constr.cpp: In function 'int main()':
test_constr.cpp:12:9: error: no matching function for call to
'point::point()'
   point p2;
         ^~
test_constr.cpp:6:3: note: candidate: 'point::point(int, int)'
   point(int x, int y) {
   ^~~~~
test_constr.cpp:6:3: note: candidate expects 2 arguments, 0 provided
```

Question. What is the meaning of this message?

If needed, two solutions enable to call the constructor without parameters: use **default values** for the parameters, or use **2 (or more) constructors**, for example

```
class point {
  ...
  public :
    point(int x=0, int y=0) {   // with default values
       x1 = x;  x2 = y;
    }
    ...
};
```

```
class point {
  ...
  public :
    point(int x, int y) {   // constructor 1
       x1 = x;  x2 = y;
    }
    point() {               // constructor 2
       x1 = 0;  x2 = 0;
    }
    ...
};
```


Exercise 1.

- Define a class *Date*, with 3 integer attributes (day, month, year). Create a constructor with parameters (do not check the relevance of their values) and a method *print_date* that enables to print the date.
- Define also a method *happy_birthday* that receives as parameters a name (C-like character string) *n* and a date *b* (the birthday of *n*) and that whishes an happy birthday to *n* if the date equals his/her birthday.

I.3 Encapsulation principle

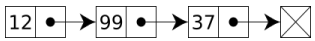
✓ Remember that *attributes* are generally *private* members, which means that values of data are hidden inside each class instance. The outside world does not know anything about the internal implementation, and communicates with the object by *message passing*, using *public* methods (for example in section I.1.2, *c1.increment();*)

Let us discuss the importance of this ***encapsulation principle*** with the example below that illustrates the relationship between data abstraction and encapsulation: *even if the structure of the internal data (attributes) changes, the outside world communicates with the object* ***in the same way***.

Example :

```
// Version 1. Class Set of integers encoded with an array
class Set {
  private :
    int *set;  // implementation with an array
    int current;
  public :
    ...
    int get_val(int i){  // method to access to element i
      return set[i-1];
    }
};
```

```
// Version 2. Class Set of integers encoded with a linked list
class Set {
  private :
    Node *first;  // implementation with a linked list
  public :
    ...
    int get_val(int i){  // method to access to element i
      Node *p = first;
      for (int j=1; j<i; j++)
         p = p->next;
      return p->value;
    }
};
```



Versions 1 and 2 offer the **same interface** (public methods), in particuler get_val (same argument type, same return value type) => whatever the implementation of `Set` is (version 1 or 2), **the application program below will be unchanged:**

```
int main() {
  Set e1;
  ...
  cout << "Element " << i << " of e1: "
       << e1.get_val(i) << endl ;
  return 0;
}
```

→ See a more comprehensive discussion on this concept here:
https://medium.com/swlh/the-importance-of-code-encapsulation-ce19efbcfe57

✓ It is also possible to define ***private methods*** in addition to the public methods which constitute the *interface* of the class.
See the example on pages 2-3. Let us also illustrate the interest of this concept with the fictional example below:

```
class Set {
  private :
    int *set;
    int current;
    int key;
    bool access_granted(int i) {  // access granted or not.
                 // this method is private, it cannot be used
                 // by the outside world
      ...
      ... // the variable key is used in this definition
    }
```

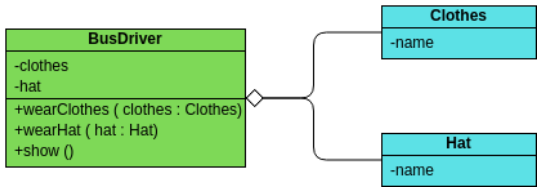


```
  public :
    ...
    int get_val(int i){  // accessor to element i
       if (access_granted(i-1))  // call to private method
         return set[i-1];
       else {
         cerr << "you cannot access this element\n";
         return -1;
       }
};

int main() {
  Set e1;
  int p;
  ...
  p = e1.get_val(i);
  if (p != -1)
     cout << "Element " << i << " of e1: " << p << endl ;
  return 0;
}
```

The accessor function `get_val` uses the private method `access_granted` in a transparent way. But the outside world **cannot use it** directly.

I.4 Object aggregation

Aggregation is the process of creating complex objects from simpler ones. In other words, class *attributes may themselves be instances of other classes*, for example:



https://online.visual-paradigm.com/fr/diagrams/templates/class-diagram/class-diagram-aggregation-example/

Strictly speaking, in the *aggregation* context, a member can belong to more than one object (class) at a time. When an *aggregation* is created, it is not responsible for creating the parts, and when it is destroyed, it is not responsible for destroying the parts (we will see "destructors" later).

The denomination "*composition*" is generally used when the whole object is res-

ponsible for the existence of its members, it is responsible for creating them when it is created, and responsible for destroying them when it is destroyed.
In that case, it is important to take care of the *initialization* and of the *destruction* of the inner objects.

To ***initialize inner objects***, the corresponding arguments to be used by their constructor must be specified in the prototype of the constructor of the outer class, after the symbol **:**, in any order and separated by commas. The constructors will be executed in the order that corresponds to the declaration of the attributes.

Example :

```
class inner_class {
  private :
     int x;
  public :
    inner_class(int z) {
       x = z;
    }
    void write() {
       cout << "x = " << x << endl;
    }
};

class outer_class {
  private :
    float y;
    inner_class i1, i2;
  public :
    outer_class(float z, int z1, int z2);
    void write() {
       cout << "y = " << y << endl;
       cout << "Object i1: \n";
       write_inner(i1);
       cout << "Object i2: \n";
       write_inner(i2);
    }
    void write_inner(inner_class i) {
       i.write();
    }
};

outer_class::outer_class(float z,int z1,int z2):i1(z1),i2(z2){
   y = z;
}

int main() {
  outer_class obj(5.8,10,15);
```



```
  obj.write();
  ...
}
```

Important note. If nothing is specified for a given inner object, its constructor must tolerate being called without argument!

Add-on. Self reference in classes

In general, within a member function, function calls and variables implicitly refer to member functions and to attributes of the current object. If it is necessary to *explicitly* refer to the object that the currently running code is part of, the keyword **this** is used. It represents a *pointer to this object*.

Exercise 2.

- Define a class *Trip*, with 3 attributes: beginning and end dates, and price (float). Define a constructor.
- Define also a method *price_per_day* that computes the price per day of the trip, and a method *print_trip* to print the characteristics of the trip.

I.5 More on constructors - destructors

I.5.0 Some preliminaries

✓ **Memory allocation and deallocation.**

The ***operators* new *and* delete** enable to dynamically allocate and deallocate memory (they represent an alternative to functions `malloc` and `free` of C)[1].

The operator **new** is followed by a type identifier (predefined or not), and the value of the corresponding expression is the *address of the allocated memory block*. Its size is sufficient to contain a value of the given type (or 0 if the allocation failed).
An array can be allocated if the type identifier is followed by an integer enclosed in `[]`.

[1] See also https://www.tutorialspoint.com/cplusplus/cpp_dynamic_memory.htm

Examples :

```
int *i = new int;     // allocation for one element
*i = 15;
...
int *x = new int[100];   // allocation for an array
        // similar to malloc(100*sizeof(int))
```

The operator **`delete`** is followed by a variable of type pointer (i.e., an address). It *deallocates the memory block* at this address. The use of `[]` is necessary to deallocate an array.

Beware, `delete` can only be used in conjunction with `new`.

Examples :

```
delete i;
delete [] x;
```

Note. For other kinds of composite types (e.g., linked lists), `delete` must be applied iteratively or recursively to deallocate the complete collection.

✓ **C++ references.**

A C++ ***reference*** (denoted using a `&`) is an alias i.e., another name for an already existing variable (it refers to an existing identifier). It means that either the variable name or the reference name may be used to refer to the variable, thus the reference can be used to *modify* the contents of the corresponding variable.

This notion is particularly useful to *pass arguments by reference*, to have the function modify the argument passed in.

Example :

```
void addOne(int &y){
  y = y + 1;
};

int main() {
  int x = 5;
  cout << "x = " << x << '\n';
  addOne(x);
  cout << "x = " << x << '\n';
  return 0;
}
```

It can also be used to *avoid making a copy of the argument* into the function

parameter, for example when passing a large struct or class to a function. In that case, the keyword **`const`** must be present, to specify that the parameter should not be modified.

Example :

```
void display(const GraphicalComponent &g){
  ...
};
```

We will come back to pass-by-reference later.

I.5.1 More on constructors, destructors, dynamic instances

✓ ***Constructors*** are methods that have the same name as their class, and that give the possibility to automatically **initialize the attributes** when an instance is created (and to perform dynamic allocation if needed).

Destructors are methods that enable **automatic memory deallocation** when the program execution leaves a block in which an instance is localized, or when an instance is explicitly "destroyed". The name of a destructor is ~ followed by the name of the class.

Note. In case of object aggregation (section I.4), the destructor of the outer class is executed after the ones of the inner objects.

Example : Counter with an identifier (character string, dynamically allocated)

```
class counter {
  private :
    char *name;
    unsigned int value;
  public :
    counter(char *n, unsigned int v) {
       name = new char[strlen(n)+1];
       strcpy(name,n);
       value = v;
    }
    void increment() {
       value++;
    }
    void decrement() {
       value--;
    }
```

```
    unsigned int get_value() {
       return value;
    }
    ~counter() {    // destructor
       delete [] name;    // deallocation
    }
};
```

And we have the following program:

```
void function_with_counter(char *name) {
  counter c1(name,0);
  for (int i=1; i<=15; i++) {
     c1.increment();
     cout << "Value of c1: " << c1.get_value() << endl;
  };
}

int main() {
  char count_name[20];
  cout << "Which name ?\n";
  cin >> count_name;
  function_with_counter(count_name);
  return 0;
}
```

→ when program execution leaves the function `function_with_counter`, the local variable `c1` is "destroyed". Then, thanks to the destructor of class `counter`, the memory block allocated for the identifier in `c1` is deallocated.

Exercise 3.

Schematically explain with simple pictures what is going on in the memory (stack, heap) when function `function_with_counter` is called and when it returns.

✓ The ***creation of a dynamic instance*** (pointer) is made using the operator **new**, and the actual parameters for the constructor are given *between parentheses after the type identifier*.

Note. For dynamic instances, message passing uses the **->** operator.

Example :

```
int main () {
   counter *c1 = new counter(5);
   for (int i=1; i<=15; i++){
      c1->increment();
```



```
      cout << "c1 contains: " << c1->get_value() << endl;
   }
   return 0;
}
```

Exercise 4.

Let us consider again the class *Date* of exercise 1, which has 3 integer attributes (day, month, year), a constructor with parameters and a method *print_date*. Write a simple *main* function that declares a dynamic instance of Date and prints it.

I.5.2 Copy constructor

✓ A ***copy constructor*** is a special kind of constructor, its prototype is of the form:

```
name-of-the-class(const name-of-the-class& parameter);
```

Example :

```
counter(const counter& c);
```

In particular, this constructor is **automatically called** when a new class instance is built from an existing instance (declaration of an instance with initialization, pass-by-value parameter, value returned by a function).

Example 1 :

```
class MyString {
  private :
    char *s;
    int len;
  public :
    MyString(const MyString &t) {
       s = new char[strlen(t.s)+1];
       strcpy(s,t.s);
       len = t.len;
    }
    // Exercise: write its "classical" constructor
    ...
};
```

Example 2 :

```
class Point {
  private:
    int x, y;
  public:
    Point(int x1, int y1){
      x = x1;
```

```
      y = y1;
    }
   int getX() { return x; }
   int getY() { return y; }
   ...
};

int main(){
  Point p1(10, 15);  // normal constructor is called here
  Point p2 = p1;   // (default) copy constructor called here
  cout << "p1.x = " << p1.getX()
       << ", p1.y = " << p1.getY();
  cout << "\np2.x = " << p2.getX()
       << ", p2.y = " << p2.getY();
  ...
  return 0;
}
```

/!\ Do not forget to define a copy constructor (and an assignment overloading - see section II) when the class contains ***dynamic data*** *(pointers)*, in order to perform deep copies instead of shallow copies, as realized by the default copy constructor.

Note. To create a (default) copy-constructor for a class that uses composition (and inheritance, see later), the compiler recursively calls the copy-constructors for all the member objects (and base classes). That is, if the member object also contains another object, its copy-constructor is also called.

Exercise 5.

We consider the following class definition:

```
class A {
  float *x;
public:
  A(float y){
    x = new float(y);
  }
  void set(float a){
    *x = a;
  }
  void print(){
    std::cout << *x << "\n";
  }
};
```

- What will be the outcome of the execution of the program below, and why?


```
int main() {
  A a1(45.78);
  A a2(a1);
  a1.print();
  a2.set(56.98);
  a1.print();
  return 0;
}
```

- What should we modify in class `A` to solve this problem?

✓ Remark. C++11 also defines the notion of *move constructor*: its goal is to "move ownership" of the attributes. It is used if the argument for construction is an r-value (a temporary object, for example the value returned by a function).

Example :

```
class MyString {
  private :
    char *s;
  public :
    MyString(MyString &&t) {
                          // t is a non-const r-value reference
       s = t.s;   // "transfers ownership" of t.s to s
       t.s = nullptr;  // sets the source pointer t.s to null
    }
    ...
};
```

Reminder. ***Self reference*** in classes: in general, within a member function, function calls and variables implicitly refer to member functions and to attributes of the current object. If it is necessary to *explicitly* refer to the object that the currently running code is part of, **`this`** represents a *pointer to this object*.

Exercise 6.

- Define a class *Array_of_float* that has as attributes: a dynamic array of float, and an integer that represents the length of this array. Create a constructor that takes as arguments an integer *len* and a float *x*, that allocates memory for an array of length *len*, and initializes the array with *x* as value for every element.
- Create a copy constructor and a destructor for this class.

- Write an independent function *display_array* that takes as argument an instance of class *Array_of_float* and that displays its contents (do you need getters in *Array_of_float*?)

If the copy constructor and the destructor print messages on the screen (for instance "Calling copy constructor" and "Calling destructor" respectively), how many messages will be printed when executing the following *main* function? Why?

```
int main(){
  Array_of_float array1(5, 3.14);
  display_array(array1);
  return 0;
}
```

I.6 Pass-by-reference parameters - const

I.6.1 Pass-by-reference parameters

✓ Remember that, in C, only pass-by-value is possible. In C++, ***pass-by-value*** and ***pass-by-reference*** are possible.

With pass-by-reference, actual parameters are *references to the storage locations of the original arguments* passed in. Thus no copy is made, and overhead of copying (time, storage in memory) is saved. Be careful, changes to these variables in the function will affect the original variables.

For a reference parameter, the unary operator **&** follows the type identifier. Inside the function, the parameter is used as in the pass-by-value case.

Example :

```
void switch_function(int &x, int &y) {
    int z = x;
    x = y;
    y = z;
}
...

int a=8,b=34;
switch_function(a,b);
```

✓ Pass-by-reference can also be used just to *save the overhead of copying the variable*. The keyword **const** can be used if we want to avoid modifying the

original value of the variable. Then the *compiler verifies* that no change is possible.

Example :

```
void format(big_structure &s) {  // const is missing
  ...
}
void print(const big_structure &s) {
   ...
   format(s);
}
```

In that case, such an error message can be obtained when compiling:

```
In function `void print(const big_structure&)':
could not convert `s' to `big_structure&'
in passing argument 1 of
`void format(big_structure&)'
```

Question. Why do we get this message?

I.6.2 Member functions and const qualifier

Let us consider the **const** qualifier in the case of ***member functions***. Here too, if const is used for a reference parameter, the compiler verifies that the variable cannot be modified by the function.

Example :

```
class MyString  {
  private :
    char str[20];
    int lg;
  public :
    MyString(char *s) {
      strcpy(str,s);
    }
    char *getstr() {
      return str;
    }
    int length() {
      return lg=strlen(str);
    }
};

void printlg(const MyString &c) {
  cout << "The length of the string is "
       << c.length() << endl;
}
```

```
int main() {
  MyString thestring("hello");
  printlg(thestring);
  return 0;
}
```

Here the function `printlg` gets as pass-by-reference parameter an instance of class `MyString`, with the qualifier `const`, which means that this instance of `MyString` cannot be modified by the function. But this constraint is not satisfied here, the compiler will print an error message of the form

```
In function `void printlg(const MyString  &)':
passing `const MyString ' as `this' argument of
`int MyString::length()' discards qualifiers
```

It means that the member function `length` ***must also declare that it does not modify the object*** (i.e. its attributes): this **`const`** declaration is put after the prototype of the function. But in that case of course, the function should not include an assignment that updates an attribute! Thus it is not sufficient to write:

```
int length() const {
  return lg=strlen(str);
}
```

that will produce the following error message:

```
In member function `int MyString::length() const':
assignment of data-member `MyString::lg' in read-only structure
```

Finally here is an acceptable solution for this example:

```
int length() const {
  return strlen(str);
}
```

Exercise 7.

- Modify the independent function *display_array* of exercise 6 so that it uses pass-by-reference for its parameter. Do you need to modify something in class Array_of_float? Why?

- If the copy constructor and the destructor print messages on the screen (for instance "Calling copy constructor" and "Calling destructor" respectively), how many messages will be printed when executing the following *main* function? Why?

```
int main(){
  Array_of_float array1(5,3.14);
  display_array(array1);
  return 0;
}
```

Add-on. In any context (constructor or other function), the **`assert`** macro can be used to check whether an assertion holds (otherwise the execution aborts).

Example :

```
class MyString  {
  private :
    char str[20];
    int lg;
  public :
    MyString(char *s) {
      assert (s != NULL);
      strcpy(str,s);
    }
   ...
};
```

```
int main() {
  char *s1 = NULL;
  MyString thestring(s1);
  ...
  return 0;
}
```

that will result into something like:

```
Assertion failed: (s != NULL), function MyString, file F1.cpp.
Abort trap: 6
```

II. OPERATOR OVERLOADING

II.0 Preamble – friend declaration

Remember that *private members* can be accessed neither from member functions of other classes nor from independent functions. C++ gives the possibility to relax this constraint, through the ***friend declaration*** (but *use it only when there is an absolute necessity!*). It enables other classes, or functions, to access (read, modify) private data.

For example, if objects of class `A` must frequently interact (using a method `f`) with objects of class `B`, it might be interesting to declare that this method `f` of class `B` is a friend of class `A`. Use this possibility exceptionally!

We will see that this notion is often used in conjunction with operator overloading.

A friend declaration can be put anywhere in a class (either in the private section or in the public section).

Example :

```
class A {
   ...
   friend int f1(float z);
                  // the independent function f1 is a friend
   friend char *B::method1();
                  // method method1 of class B is a friend
   friend class C;  // class C is a friend
   ...
}
```

Here the function `f1`, the method `method1` of class `B`, and all the methods of class `C` are allowed to access private members of class `A`.

II.1 Operator overloading

II.1.1 General notions - motivations

C++ enables *to **overload** most of its operators*, which means that their definition is **extended to deal with new data types** (new classes).

Example :

Let us assume that a specific class has been defined to represent *complex numbers*, and that this class proposes a method *add* to perform the addition of two complex numbers.

We recall that a *method call corresponds to message passing*, which means that this function receives 1 parameter, and can be called as follows (where `z1` and `z2` are two instances of the class):

```
z1.add(z2);   ← the message "add" is sent to z1 with z2 as parameter
```

or

```
z2.add(z1);   ← the message "add" is sent to z2 with z1 as parameter
```

In contrast, *overloading the definition of the + operator* for objects of this class will enable the use of expressions such as:

```
z1 + z2      ← the + operator is overloaded for the class of complex numbers
```

II.1.2 Operator overloading technique

Note that overloaded operators keep their usual *priority* and *associativity* (they must also have the same "arity"). At least one of their arguments *must be a class instance*.

✓ An ***overloaded binary operator*** can be defined:

- either by a ***method*** of the class, that has only *one* argument. The argument "this" is implicitly present.
- or by an ***independent function***, that has *two* arguments (and that is usually declared as a friend).

Similarly, an ***overloaded unary operator*** can be defined:

- either by a *method* of the class, that has *no* argument
- or by an *independent function*, that has *one* argument (and that is usually declared as a friend).

The notation : ***<operand$_1$> op <operand$_2$>***

is equivalent to: ***<operand$_1$>.op(<operand$_2$>)***

if *op* is overloaded as a member function,

and to: *op(<operand$_1$>,<operand$_2$>)*

if *op* is overloaded as an independent function.

Beware, the overloading for the operators ++ and -- is in fact the overloading of their "pre" version: an additional (and useless) argument, of type int, must be used to perform the overloading of the "post" version.

Example :

```
class counter {
  private :
    unsigned int value;
  public :
    counter(unsigned int v=0) {
       value = v;
    }
    unsigned int operator++();     // prefix version
    unsigned int operator--(int);  // postfix version
    unsigned int operator()();
    unsigned int operator+(counter);
    unsigned int operator+(unsigned int);
};

unsigned int counter::operator++() {
  if (value<65535) return ++value;
  else return value;
}

unsigned int counter::operator--(int i) {
  if (value>0) return value--;
  else return value;
}

unsigned int counter::operator()() {
  return value;
}

unsigned int counter::operator+(counter a) {
  return (value += a.value);
}

unsigned int counter::operator+(unsigned int x) {
  return (value += x);
}

int main() {
  counter my_counter;
  counter c2(50);
  unsigned int sum;
  for (int i = 0; i<12; i++) {
     ++my_counter;
```



```
     cout << "current value of my_counter = "
          << my_counter() << endl;
  }
  sum = my_counter + c2;
  cout << "New value of my_counter = "
       << my_counter() << endl;
  sum = c2 + my_counter--;
  cout << "New value of my_counter = "
       << my_counter() << endl;
  cout << "... and sum = " << sum << endl ;
  ...
}
```

The execution of this program gives:

```
current value of my_counter = 1
current value of my_counter = 2
current value of my_counter = 3
current value of my_counter = 4
current value of my_counter = 5
current value of my_counter = 6
current value of my_counter = 7
current value of my_counter = 8
current value of my_counter = 9
current value of my_counter = 10
current value of my_counter = 11
current value of my_counter = 12
New value of my_counter = 62
New value of my_counter = 61
... and sum = 112
```

Now let us see the same example *with (friend) independent functions*:

```
class counter {
  friend unsigned int operator++(counter &);
  friend unsigned int operator--(counter &, int);
  friend unsigned int operator+(counter &, counter);
  friend unsigned int operator+(counter &, unsigned int);
  private :
    unsigned int value;
  public :
    counter(unsigned int v=0) {
       value = v;
    }
  unsigned int operator()();
 };

unsigned int counter::operator()() {
  return value;
}

unsigned int operator++(counter &c) {
  if (c.value<65535)
    return ++(c.value);
```

```
  else return c.value;
}

unsigned int operator--(counter &c, int i) {
  if (c.value>0)
    return (c.value)--;
  else return c.value;
}

unsigned int operator+(counter &c, counter a) {
  return (c.value += a.value);
}

unsigned int operator+(counter &c, unsigned int x) {
  return (c.value += x);
}
```

Of course the associated `main` is unchanged, and the result is the same.

✓ All the following **operators *can be overloaded***:

- Binary operators :

  = () [] -> new delete *(must be overloaded as member functions)*
  * / % + - << >> < <= > == != & ^ || && | += -= *= /= %= &= ^= |= <<= >>= ,

- Unary operators :

  + - ++ -- ! ~ * & (cast)

But the following ones ***cannot be overloaded***:

:: . ?: sizeof

Exercise 8.

- Define a class *Complex*, with two attributes *real* and *imaginary* that are floating point numbers, and define a constructor
- Define a method *print_complex* that displays the complex number
- Overload the + operator for this class, as a method, and as an independent function (it returns a new *Complex* which is the sum of two *Complex*)
- Write a simple *main* function that declares two complex numbers, computes and prints their sum.

II.2 Overloading for the iostream operators

The operators **<<** and **>>** are very frequently overloaded. This is possible only with independent functions.

The prototypes of these functions are:

```
ostream &operator<<(ostream &o, const class_name &);
istream &operator>>(istream &i, class_name &);
```

Example :

```
class Complex {
  friend ostream &operator<<(ostream &, const Complex &);
  friend istream &operator>>(istream &, Complex &);
  private :
    float real, imaginary;
  public :
    ...
};

ostream &operator<<(ostream &o, const Complex &c) {
  o << "Real part: " << c.real
    << " , imaginary part: " << c.imaginary << "\n";
  return o;
}

istream &operator>>(istream &i, Complex &c) {
  cout << "Real part? \n";
  i >> c.real;
  cout << "Imaginary part? \n";
  i >> c.imaginary;
  return i;
}
```

Exercise 9.

- Add the following operators to class *Date* of exercise 1: an overloading of the << operator, and an overloading of the prefix ++ operator (it creates a new *Date* with the same day, the same month, and an incremented year)
- Adapt the *main* function consequently.

II.3 Overloading of the assignment operator

In section I.5, we have discussed the definition of the copy constructor. The purpose of the ***assignment operator* overloading** is almost equivalent (though the

copy constructor is used to initialize *new* objects whereas the assignment operator overwrites the contents of *existing* objects).

As for the copy constructor, it is preferable to overload the assignment operator when the class contains *dynamic data (pointers)*.

The definition of the assignment operator *should detect self-assignment* (to prevent unexpected effects e.g., when the class contains a pointer and the operator releases the pointed object before trying to assign it).

Example :

```
class MyString {
  private :
    char *s;
    int len;
  public :
    MyString& operator=(const MyString &t) {      // returns an
                                  // implicit pointer to its return value
       if (this != &t) {
         delete [] s;
         s = new char[strlen(t.s)+1];
         strcpy(s,t.s);
         len = t.len;
       }
       return *this;
    }
    ...
};
```

Note. As for the copy constructor, a default definition of `operator=` (with memberwise assignments) is automatically created by the compiler if needed.

III. GENERIC CLASSES

In section IV, we will describe some components of the STL library. Some of these components are *generic classes (template)*, let us present this concept.

The concept of ***template*** enables the definition of *models of functions or classes* (i.e., they model a given template).

✓ A generic class is defined as follows:

```
template <class Type>
class Class_name {
   ...
   type1 method1(...);
   type2 method2(...);
   ...
};

template <class Type>
type1 Class_name<Type>::method1(...) {
  ...
}

template <class Type>
type2 Class_name<Type>::method2(...) {
  ...
}
```

`Type` can be used ***everywhere*** in the class definition (attribute, function parameters or return value,…).

The definition of a member function *outside* the class declaration must be preceded by the specification: `template <class Type>`,

and the full name of each member function is of the form:

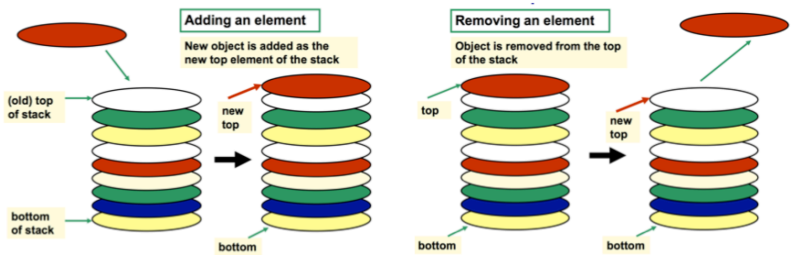
```
Class_name<Type>::Method_name
```

where `Type` is the identifier used in the *template* clause.

✓ Then, this `Type` will be instantiated when *declaring* ***class instances***, for example:

```
Class_name<int> var1;
Class_name<char> var2;
...
```

Example : let us illustrate these concepts with a class `Stack` that represents a stack of elements of any type `Type`



http://slideplayer.com/slide/6976797/

```
template <class Type>
class Stack;

template <class Type>
 ostream &operator<<(ostream &o, const Stack<Type> &p) {
   o << "Elements of the stack: " ;
   for (int i=0; i<=p.top ; i++)
     o << p.the_stack[i] << " " ;
   o << endl;
   return o;
 }

template <class Type>
class Stack {
  // note the specific syntax for friend declarations:
  friend ostream &operator<< <>(ostream &o,
                                const Stack<Type> &);
  private :
    Type *the_stack;
    int top;
    const int MAX_STACK;
  public :
    Stack();
    Stack(int);
    ~Stack();
    void push(Type);
    void pop();
    Type first_elt();
    bool empty();
 };

 template <class Type>
 Stack<Type>::Stack() : MAX_STACK(50) {
   the_stack = new Type[MAX_STACK];
   top = -1;
 }
```



```
template <class Type>
Stack<Type>::Stack(int max) : MAX_STACK(max) {
  the_stack = new Type[MAX_STACK];
  top = -1;
}

template <class Type>
Stack<Type>::~Stack() {
  delete [] the_stack;
}

template <class Type>
void Stack<Type>::push(Type elt) {
  top++;
  if (top < MAX_STACK)
    the_stack[top] = elt;
  else cerr << "The stack is full!\n";
}

template <class Type>
void Stack<Type>::pop() {
  if (top >= 0) top--;
}

template <class Type>
Type Stack<Type>::first_elt() {
  return the_stack[top];
}

template <class Type>
bool Stack<Type>::empty() {
  return (top == -1) ? true : false;
}

int main() {
  Stack<int> p1, p2(5);
  for (int i=0; i<5; i++)
    p1.push(i);
  while (!p1.empty()) {
    p2.push(p1.first_elt());
    cout << "This element is pushed on p2: "
         << p2.first_elt() << "\n";
    p1.pop();
  }
  cout << p2;
  return 0;
}
```

The execution of this program gives:

```
This element is pushed on p2: 4
This element is pushed on p2: 3
This element is pushed on p2: 2
This element is pushed on p2: 1
This element is pushed on p2: 0
Elements of the stack: 4 3 2 1 0
```

✓ Note that ***independent functions*** can also be ***generic***. When the function is called, if the actual value of the parameter type can be determined from the types of the function parameters, its explicit instantiation can be omitted.

Example :

```
template <class T>
T scan_and_print() {
  T data;
  cout << "Give the expected data\n";
  cin >> data;
  cout << "Data: " << data << endl;
  return data;
}

template <class T1, class T2>
void print_data(T1 data1, T2 data2) {
  cout << "Data1 and data2 : " << data1
       << " and " << data2 << endl;
}

int main() {
  int x;
  x = scan_and_print<int>();
  float f = 9.34;
  print_data(x,f);
  return 0;
}
```

✓ Finally let us also note that function or class templates *can also* ***use arithmetic parameters***, for instance *integers*.

Example : array of `float` with indices between A and B

```
template<int A, int B>
class Array {
    float the_array[B-A+1];
  public :
    float operator[](int i);
    ...
};

template<int A, int B>
float Array<A,B>::operator[](int i) {
  if (i>=A and i<=B)
    return the_array[i-A];
  else {
       cerr << i << " index out of bounds "
            << A << " and " << B << endl;
       return 0;
  }
}
```


Exercise 10. (optional)

We assume that three classes *Dollar*, *Euro* and *Pound* have been defined to represent respectively an amount of money in dollars, euros, and pounds. Each class has: a constructor that takes as parameter a *float* to initialize the amount, a method *get_value()* that returns the amount, and an overloading of the << operator. We also have the following function that checks whether the datatype of its parameter is *Dollar*, *Euro* or *Pound*:

```
#include <typeinfo>

template <class Type_currency>
bool check_type(Type_currency object){
  return typeid(object) == typeid(Dollar)
         || typeid(object) == typeid(Euro);
         || typeid(object) == typeid(Pound);
}
```

Define a template class *Bank_account*, generic on a type *Type_currency*, that has two attributes: a character string *owner_name*, and a *Type_currency balance*. This class has a constructor, a copy constructor, a destructor, and a method *credit_balance()* that returns true if the amount of *balance* is positive, and it can be used for example as follows:

```
int main() {
  Dollar d(100.5);
  Euro e(50);
  if (check_type(d) && check_type(e)) {
    Bank_account<Dollar> b1("Paul", d);
    if (b1.credit_balance())
      cout << "This account has a credit balance \n";
    else cout << "This account has a debit balance \n";
    Bank_account<Euro> b2("Mark", e);
    if (b2.credit_balance())
      cout << "This account has a credit balance \n";
    else cout << "This account has a debit balance \n";
  }
  return 0;
}
```

Define an overloading of the << operator for this class.

IV. STANDARD TEMPLATE LIBRARY

IV.1 Overview of the STL - Containers

The C++ standard library, **STL** (Standard Template Library, see for instance http://www.cplusplus.com/reference/stl/), contains classes and functions for:

- containers (vectors, lists, stacks, sets,...) and associated algorithms,
- iterators, to iterate inside containers,
- inputs/outputs (as already mentioned in section I.1.3),
- some other constructs, like character strings.

Note. All these elements are in the standard namespace `std`, thus they will be used with the prefix `std::` or after declaring:

```
using namespace std;
```

IV.1.1 Containers

✓ ***Containers*** enable to represent and to use *collections of objects*. Various containers are available: vectors, lists, stacks, sets,… All these classes are defined such that they can contain objects of any type, they are *generic*. Beware, it means that every instantiation of a container must specify the type of its objects.

✓ Let us give an overview of the ***vectors***, represented by the class `vector` (it is necessary to include `<vector>`). See http://www.cplusplus.com/reference/vector/vector/. The notion of vector is close to the notion of array, but the main advantage is that the size of a vector dynamically changes according to the number of elements inserted in the vector.

Among the most useful methods:

- `size` : returns the number of elements in the vector
- `empty` : returns whether the vector is empty
- `push_back` : adds a new element at the end of the vector, after its current last element
- `insert` : inserts a new element in the vector, at a given position.

- `begin` and `end` : return *iterators* that point respectively to the first element and to the position just after the last element. They can be used to perform an iterative traversal of the vector (though a classical iteration, using indices, is also possible).

Note also that a vector can be assigned the value of another vector (operator **=**), can be compared to another vector (operator ==), and that it is possible to get the value of an element of a vector by means of the **[ ]** operator.

IV.1.2 Iterators

The ***iterators*** (`iterator`) are a kind of generalization of pointers. They represent positions of elements, and they exist for every type of container. The type of the iterator depends on the type of the associated container. The specific type `const_iterator` enables to specify an iterator that is used to access elements of a container *without modifying them*.

Various operators can be used with iterators e.g., `*` refers to the container element that stands at the position specified by the iterator, and `++` and `--` enable to increment and to decrement the value of the iterator.

Example :

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

using namespace std;

int main() {
  const int the_size = 10;

  // The vector is filled with 10 elements
  vector<float> myvector(the_size);
  cout << "Give the elements of the vector" << endl;
  for (vector<float>::iterator i = myvector.begin();
                                       i!=myvector.end(); i++)
    cin >> *i;

  // The content of the vector is displayed
  cout << "Here is the content of the vector: " << endl;
  for (vector<float>::const_iterator
              i = myvector.cbegin(); i!=myvector.cend(); i++)
    cout << *i << " ";
  cout << endl;
```

```
  // Creation of a copy (illustration of =)
  vector<float> copyvect;
  copyvect = myvector;
  copyvect.push_back(111.2);  // add an element

  // The content of copyvect is displayed
  cout << "Content of the new vector: " << endl;
  for (vector<float>::const_iterator
            i = copyvect.cbegin(); i!=copyvect.cend(); i++)
    cout << *i << " ";
  cout << endl;
  return 0;
}
```

Here is an example execution:

```
Give the elements of the vector
8 5 3.8 12 89 2.1 9 -0.7 6 23
Here is the content of the vector:
8 5 3.8 12 89 2.1 9 -0.7 6 23
Content of the new vector:
8 5 3.8 12 89 2.1 9 -0.7 6 23 111.2
```

Remark. C++11 also supports a more compact solution, under the form of range-based *for* loops (in that case, compile with the `-std=c++11` option) :

```
int main() {
  const int the_size = 10;

  // The vector is filled with 10 elements
  vector<float> myvector(the_size);
  cout << "Give the elements of the vector" << endl;
  for (float &v : myvector)
    cin >> v;

  // The content of the vector is printed
  cout << "Here is the content of the vector: " << endl;
  for (float v : myvector)
    cout << v << " ";
  cout << endl;

  // Creation of a copy (illustration of =)
  vector<float> copyvect;
  copyvect = myvector;
  copyvect.push_back(111.2);  // add an element

  // The content of copyvect is printed
  cout << "Content of the new vector: " << endl;
  for (float v : copyvect)
    cout << v << " ";
  cout << endl;
  return 0;
}
```


IV.2 Algorithms

✓ The STL also proposes ***algorithms*** (see http://www.cplusplus.com/reference/algorithm/) that can be applied to (parts of) containers.

For example, the content of a vector (or of most of the other containers) can be ***sorted***, using the function `sort`. The elements are sorted in ascending order. Be careful: the operator < must be defined for the type of the vector elements.
To sort the full content of a vector `v`, the interval of elements to be sorted is from the beginning to the end, hence you must call `sort(v.begin(), v.end())`.
Note that there exists a version of `sort` that can receive as third parameter the *address of a comparison function*, thus allowing to use another sorting policy than the ascending order.

There exists various other algorithms, for example the function `reverse` that enables to ***reverse*** the order of the vector elements, inside a given interval.

Example :

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

using namespace std;

int main() {
  const int the_size = 10;

  // The vector is filled with 10 elements
  vector<float> myvector(the_size);
  cout << "Give the elements of the vector" << endl;
  for (vector<float>::iterator i = myvector.begin();
                                   i!=myvector.end(); i++)
    cin >> *i;

  // The vector is sorted:
  sort(myvector.begin(), myvector.end());
  cout << "Content after sorting: " << endl;
  for (vector<float>::const_iterator
            i = myvector.cbegin(); i!=myvector.cend(); i++)
    cout << *i << " ";
  cout << endl;

  // Creation of a copy (illustration of =)
  vector<float> copyvect;
```

```
  copyvect = myvector;
  copyvect.push_back(111.2);  // add an element

  // The new vector is reversed:
  reverse(copyvect.begin(), copyvect.end());
  cout << "Content after reversing: " << endl;
  for (int i=0; i < copyvect.size(); i++)
    cout << copyvect[i] << " ";
  cout << endl;
  return 0;
}
```

Here is an example execution:

```
Give the elements of the vector
8 5 3.8 12 89 2.1 9 -0.7 6 23
Content after sorting:
-0.7 2.1 3.8 5 6 8 9 12 23 89
Content after reversing:
111.2 89 23 12 9 8 6 5 3.8 2.1 -0.7
```

✓ Beware, remember that the **operator <** must be defined for the type of the elements of the vector. This property holds for the example above, because the vector elements are of type *float*.

Let us see an **example** which **requires the definition** of this operator:

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

using namespace std;

class Point {
private:
  int x1, x2;
public :
  Point(int x, int y) {
    x1 = x;  x2 = y;
  }
  void print() const {
    cout << "( " << x1 << " , " << x2 << " )  ";
  }
  bool is_zero(){
    return x1==0 && x2==0;
  }
  bool operator<(Point p) const {
    return x1<p.x1 || (x1==p.x1 && x2<p.x2);
  }
};
```



```
int main() {
  // The vector is filled with elements of type Point
  vector<Point> myvector;
  myvector.push_back(Point(6,8));
  myvector.push_back(Point(3,-5));
  myvector.push_back(Point(1,9));
  myvector.push_back(Point(11,7));
  myvector.push_back(Point(3,8));

  // The vector is sorted:
  sort(myvector.begin(), myvector.end());
  cout << "Content after sorting: " << endl;
  for (vector<Point>::const_iterator
            i = myvector.cbegin(); i!=myvector.cend(); i++)
    i->print();
  cout << endl;
  return 0;
}
```

which results in:

```
Content after sorting:
( 1 , 9 )  ( 3 , -5 )  ( 3 , 8 )  ( 6 , 8 )  ( 11 , 7 )
```

IV.3 Character strings

✓ Character strings can be represented by the class **`string`** which provides various functions (see http://www.cplusplus.com/reference/string/string/). Do not forget to include the header `<string>`.

As for the arrays, the index of the first character is 0.

Note that the usual type `char *` is still allowed, together with the functions of the C library (`strcpy`, `strcmp`,...). In general, C++ programmers prefer the use of the `string` class.

✓ This class includes several constructors, a destructor, and various methods that enable to extract a sub-string, to compare strings, to concatenate strings, to insert or to replace characters inside a string, to find if characters belong to a string,… Some of these functions are illustrated in the example below.

Example :

```
#include <iostream>
#include <string>

using namespace std;
```

```
int main() {
  // initialization of the strings:
  string mystring1 = "Hello", mystring2("Jim");
  string mystring3(4,'!');
  string res1;

  // concatenation and length:
  res1 = mystring1 + " " + mystring2 + " " + mystring3;
  cout << "res1 = " << res1 << " and its length is "
       << res1.length() << endl;

  // extraction of a sub-string:
  string res2 = res1.substr(6,3);
  cout << "   and the sub-string, between char. 6 and 8 = "
       << res2 << endl;

  // look for characters:
  int n1, n2;
  n1 = res1.find("ll",0);
  if (n1 != -1)
    cout << "Found 'll' at position " << n1 << endl;
  else cout << "String not found... " << endl;
  n2 = res1.find_first_of("ahil",0);
  if (n2 != -1)
    cout << "One of the charact. a,h,i,l is found at position "
         << n2 << endl;
  else cout << "Characters not found... " << endl;

  // insertion :
  string res3 = res1.insert(5,1,',');
  cout << "After inserting a comma: " << res3 << endl;

  // comparison :
  if (res1.substr(0,7) > "Greetings")
    cout << res1.substr(0,5)
         << " is greater than Greetings" << endl;
  else cout << res1.substr(0,5)
            << " is lesser than Greetings" << endl;
  return 0;
}
```

which results in:

```
res1 = Hello Jim !!!! and its length is 14
   and the sub-string, between char. 6 and 8 = Jim
Found 'll' at position 2
One of the charact. a,h,i,l is found at position 2
After inserting a comma: Hello, Jim !!!!
Hello is greater than Greetings
```

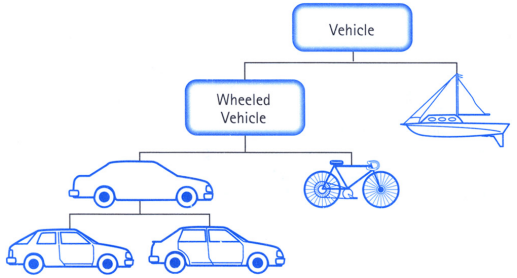
Note the overloading of the operator <<, of the operator + for the concatenation, and of the comparison operators.

V. INHERITANCE AND DERIVED CLASSES

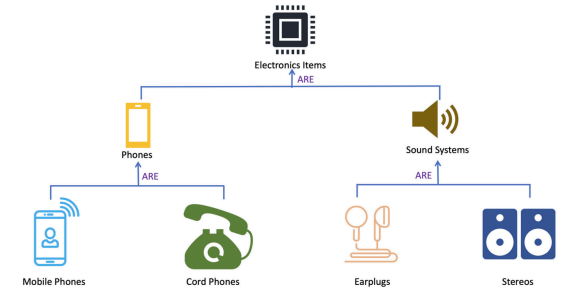
V.1 General notions

✓ The notion of ***inheritance*** enables to build a **hierarchy** of reusable software components. Subclasses are specializations of the super class (parent class).

Examples :



https://www.java67.com/2016/09/oops-concept-tutorial-in-java-object-oriented-programming.html



https://towardsdatascience.com/how-to-code-inheritance-in-java-beginners-tutorial-in-oop-d0fc0a71be98

The basis of inheritance in C++ is the construction of ***derived classes***. Note that a class can have several parent classes (which is called *multiple inheritance*, not studied here for the sake of simplicity).

✓ Each object oriented language has its own inheritance characteristics. A derived class in C++ *inherits all base class methods, it can redefine them, and it can also have additional methods*. It can also have additional attributes.

In C++, a derived class ***cannot*** *access the private data* of its base class. The

protected mode is used to give access to inherited data: a protected member can be accessed by the members of any derived class.

V.2 Definition of derived class

✓ A derived class is usually built as follows:

```
class identifier : public parent_name
{
   ...
};
```

or even

```
class identifier : parent_name
{
   ...
};
```

The first case is called *public inheritance*, and the second one is called *private inheritance*:

- *public inheritance*: public members of the base class become public members of the derived class, and protected members of the base class become protected members of the derived class,
- *private inheritance*: public and protected members of the base class become private members of the derived class.

In any case, private members of the parent class are not accessible directly from the derived class (but can be accessed through calls to public and protected methods of the base class).

Example :

```
class A {
  ...
  public :
    int fct(...);
  ...
};

class B : public A {
  ...
};
```



```
int main() {
  A obj1;
  B obj2;
  obj1.fct(...);
  obj2.fct(...);     // fct acts on obj2 as it acts on
  ...                // obj1 (type A)
}
```

✓ Note that, in any case, the **scope resolution operator** **::** enables to access members of the base class, in particular when the derived class ***redefines (overrides) an inherited method*** and when this redefinition uses the original one.

Example : Two inherited methods are redefined in this example

```
class parent_class {  // Note. No constructor in this first example
   private :
     int i1, i2;
   public :
     void assign(int p1, int p2) {
        i1 = p1;
        i2 = p2;
     }
     int inc1() {
        return ++i1;
     }
     int inc2() {
        return ++i2;
     }
     void display() {
        cout << i1 << " , " << i2;
     }
};

class derived_class : public parent_class {
   private :
     float i3;
   public :
     derived_class(float p3) {
        i3 = p3;
     }
     int inc1() {         // redefined
        cout << "redefinition of inc1\n";
        return parent_class::inc1();
     }
     void display() {     // redefined
        parent_class::display();
        cout << "i3 = " << i3 << endl;
     }
};
```

```
int main() {
   parent_class p;
   p.assign(-2,-4);
   p.display();
   derived_class d2(5.2);
   d2.assign(-6,-8);
   d2.inc1();      // which one?
   d2.inc2();      // which one?
   d2.parent_class::inc1();    // which one?
   d2.display();   // which one?
   ...
   return 0;
}
```

✓ See more details about the way member functions are selected by the compiler, for example on this page: *https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/*

Basically: When a member function is called with a derived class object, the compiler first looks to see if that member exists *in the derived class*. If not, it begins walking up the inheritance chain and checking whether the member has been *defined in any of the parent classes*. It uses the first one it finds.

V.3 Using the constructors of the parent class

✓ Since each instance of a derived class contains its copy of the inherited data, they must be initialized too. Let us see the use of the ***constructor***.

If a class B is derived from a class A, the constructor of B must pass parameters to the constructor of the base class A (unless A has no constructor, or a constructor without parameters). To that goal, *parameter passing to the base class constructor is added in the initialization list of the derived class*, after the **:** character.

Example : Let us add constructors to the previous example

```
class parent_class {
   private :
     int i1, i2;
   public :
     parent_class(int p1, int p2) {
        i1 = p1;
        i2 = p2;
     }
```



```
     int inc1(){
        return ++i1;
     }
     ...
};

class derived_class : public parent_class {
   private :
     float i3;
   public :
     derived_class(int p1, int p2, float p3) :
                                        parent_class(p1,p2) {
        i3 = p3;
     }
     int inc1() {
        cout << "redefinition of inc1\n";
        return parent_class::inc1();
     }
     ...
};

int main() {
   parent_class p(-2,-4);
   p.display();
   derived_class d1(-6,-8,5.2);
   d1.inc1();
   ...
}
```

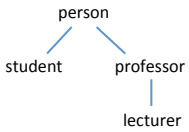
✓ ***Inheritance*** and ***aggregation*** can be used simultaneously. In that case, the constructor must pass parameters to the constructors of the base class and of the member objects.

Example :

```
class parent_class {
   private :
     int i1, i2;
   public :
     parent_class(int p1, int p2) {
        i1 = p1;
        i2 = p2;
     }
     int inc1() {
        return ++i1;
     }
     ...
};

class inner_class {
   private :
     char x;
```

```
    public :
      inner_class(char z) {
        x = z;
      }
      void write() {
        cout << "x = " << x << endl;
      }
};

class derived_class : public parent_class {
   private :
     float i3;
     inner_class in;
   public :
     derived_class(int p1, int p2, float p3, char p4) :
                              parent_class(p1,p2), in(p4) {
        i3 = p3;
     }
     int inc1() {
        cout << "redefinition of inc1\n";
        return parent_class::inc1();
     }
     ...
};

int main() {
   parent_class p(-2,-4);
   p.display();
   derived_class d1(-6,-8,5.3,'a');
   d1.inc1();
   ...
}
```

✓ **Example** : this example describes a hierarchy of persons in a university (students, employees). The class person gathers the features that are common to all the categories of persons.




```
class person {
   private :
     string name;          // Name
     string street_name;   // Address
     int number;
     string city;
   public :
     person(string n, string s, int num, string c);
     void print();
     ...
};

person::person(string n, string s, int num, string c) {
   name = n;
```



```
   street_name = s;
   number = num;
   city = c;
}

void person::print() {
   cout << "Name = " << name << "\n"
        << "Address = " << number << " "
        << street_name << " street, " << city << "\n";
}

class student : public person {
   private :
     int id_number;   // Identifier
     int level;       // Level (year)
   public :
     student(string n, string s, int num, string c,
             int i, int l);
     void print();
     ...
};

student::student(string n, string s, int num, string c,
                 int i, int l) : person(n, s, num, c) {
   id_number = i;
   level = l;
}

void student::print() {
   person::print();
   cout << "id_number = " << id_number << "\n"
        << "level = " << level << "\n";
}

class professor : public person {
   private :
     string dept;    // Department
     float salary;
   public :
     professor(string n, string s, int num, string c,
               string d, float sal);
     void print();
     ...
};

professor::professor(string n, string s, int num, string c,
                     string d, float sal) :
                                    person(n, s, num, c) {
   dept = d;
   salary = sal;
}

void professor::print() {
   person::print();
   cout << "dept = " << dept << "\n"
```

```
        << "salary = " << salary << "\n";
}

class lecturer : public professor {
   private :
     float hourly_wage;
   public :
     lecturer(string n, string s, int num, string c,
              string d, float sal, float h);
     void print();
     ...
};

lecturer::lecturer(string n, string s, int num, string c,
                   string d, float sal, float h) :
                        professor(n, s, num, c, d, sal) {
   hourly_wage = h;
}

void lecturer::print() {
   professor::print();
   cout << "hourly_wage = " << hourly_wage << "\n";
}

int main() {
   student s("Dupont", "Gambetta", 3, "Grenoble", 123, 4);
   professor p("Dubois", "Vauban", 5, "Valence",
               "CS", 2500);
   lecturer st("Martin", "Colbert", 26, "Lyon", "Maths",
               2000, 30);
   ...
   s.print();   // method "print" of the class "student"
   p.print();   // method "print" of the class "professor"
   st.print();  // method "print" of the class "lecturer"
   ...
}
```

Note. The "protected" mode has not been used here, since the methods of the derived classes *do not* have to directly access the private data of the base class.

Exercise 11.

Consider the example of class *counter* of page 27 (without operator +):

```
class counter {
  private :
    unsigned int value;
  public :
    counter(unsigned int v=0) {
       value = v;
    }
    unsigned int operator++();     // prefix version
```



```
    unsigned int operator--(int);  // postfix version
    unsigned int operator()();
};
```

and **derive** from this class a class *range_limited_counter* in which the value of the counter is limited to a *given maximum value*, given as a parameter to the constructor.

→ Which methods should be redefined? Should additional methods be defined?

Make sure that the program below is usable with this class:

```
int main() {
   counter c1;
   range_limited_counter c2(1,10);   // max value 10
   for (int i=1; i <= 15; i++) {
     ++c1;
     ++c2;
     cout << "Value of c1: " << c1()
          << " and value of c2: " << c2() << endl;
   }
   return 0;
}
```

What is the resulting execution? Which methods are used?

V.4 Compatibility

✓ Let us now consider the ***compatibility between instances*** (static or dynamic) of a base class and its derived classes.

We take as example the following configuration:

```
class A {
    ...
};

class B : public A {
    ...
};

int main() {
   ...
   A obj1;
   B obj2;
   A *p_obj1;
   B *p_obj2;
   ...
}
```

With this configuration, the following assignments can be performed:

```
p_obj1 = p_obj2;
obj1 = obj2;  // obj2 is converted into the type A
```

but this assignment is not feasible:

```
obj2 = obj1;
```

Note also that the assignment below is legal (though not relevant), with its explicit type conversion (cast):

```
p_obj2 = (B*) p_obj1;
```

✓ Finally, it is of the utmost importance to remember that, everywhere an object of type `A *` is expected as a *function parameter*, **the actual parameter can be of type `B *`** (or of its derived classes).

Exercise 12.

-1- Add to classes *counter* and *range_limited_counter* of exercise 11 a method *print* that displays the characteristics of the counter. Then write a *main* function that:

- creates a *counter c0* with initial value 0, a *range_limited_counter c1* with initial value 1 and maximum value 10, a pointer to *counter pc2* with initial value 2, and a pointer to *range_limited_counter pc3* with initial value 3 and maximum value 20
- assigns *c1* to *c0* and prints the characteristics of *c0*
- assigns *c0* to *c1* and prints the characteristics of *c1*
- assigns *pc3* to *pc2* and prints the characteristics of *pc2*
- assigns *pc2* (the original one) to *pc3* and prints the characteristics of *pc3*

Will these assignments be viable, and what will be the outcome?

-2- Define an independent function *is_null* that receives as parameter a *counter* by reference, and that returns true if the value of this counter is 0, false otherwise. Then write a *main* function that:

- creates a *counter c0* with initial value 10, a *range_limited_counter c1* with initial value 1 and maximum value 100, a pointer to *counter pc2* with initial value 0, and a pointer to *range_limited_counter pc3* with initial value 3 and maximum value 20
- declares a *vector v1* of *counter* and inserts *c0* and the counter pointed to by *pc2* in *v1*, then declares a *vector v2* of *range_limited_counter* and inserts *c1* and the counter pointed to by *pc3* in *v2*
- goes through *v1* (use a const_iterator), prints the characteristics of each element and whether its value is 0 or not, and similarly for *v2*

What will be the outcome?

-3- Add what is necessary to classes counter and/or range_limited_counter to make it possible to **sort** the contents of the vectors, modify the main function to sort the vectors prior to display their contents. What will be the outcome?

VI. POLYMORPHISM AND VIRTUAL FUNCTIONS

VI.1 Early binding and late binding

✓ C++ implements early binding and late binding (**early binding is the default**). Let us describe their characteristics:

- *early binding* : *during* ***compilation***, it is decided how a given action will be implemented (according to the *type that has been* ***declared*** for the object that receives the corresponding message),
- *late binding* : this decision is taken dynamically, ***at runtime***, according the **actual** type of the object when it receives the corresponding message.

The main advantage of the first solution is execution speed (note also that the code can be optimized by the compiler), but the second solution offers more flexibility.

✓ Reminder: in the configuration considered at the end of section V

```
class A {
    ...
};

class B : public A {
    ...
};

int main() {
   ...
   A obj1;
   B obj2;
   A *p_obj1;
   B *p_obj2;
   ...
}
```

the following assignment is feasible:

```
p_obj1 = p_obj2;
```

but note that the (declared) type of `p_obj1` is `A *`

Therefore, even after this assignment, a statement of the form `p_obj1->fct(...)` will (by default) call the function `fct` of class `A` (even if redefined in class `B`).

However `p_obj1` is actually a pointer to an object of type `B`. We will see later that

this default behaviour can be modified, if needed.

Exercise 13.

To clarify the latter point, let us come back to simple classes *counter* and *range_limited_counter* as defined below:

```
class counter {
private :
  unsigned int value;
public :
  counter(unsigned int v=0) {
    value = v;
  }
  unsigned int increment(){
    cout << "--> counter::increment\n";
    if (value<65535) return ++value;
    else return value;
  }
  unsigned int operator()() {
    return value;
  }
};

class range_limited_counter : public counter {
private :
  unsigned int max_val;   // maximum value
public :
  range_limited_counter(unsigned int val=0,
                        unsigned int max=256): counter(val) {
    max_val = max;
  }
  unsigned int increment() {
    cout << "--> range_limited_counter::increment\n";
    if ((*this)() < max_val)
      return counter::increment();
    else return (*this)();
  }
};
```

What will be the outcome of the execution of the following `main`? (i.e., what will be printed?)

```
int main() {
  counter c0;
  range_limited_counter c1(1,10);

  c0 = c1;
  cout << "c0 (after assignment of c1): ";
  c0.increment();

  counter *pc2 = new counter(2);
```

```
  range_limited_counter *pc3 = new range_limited_counter(3,20);

  pc2 = pc3;
  cout << "pc2 (after assignment of pc3): ";
  pc2->increment();

  pc2 = new counter(2);
  pc3 = (range_limited_counter *)pc2;
  cout << "pc3 (after assignment of pc2): ";
  pc3->increment();
  return 0;
}
```

✓ In C++, it is possible to implement *late binding* using the concept of ***virtual functions***, thus concretizing polymorphism. This concept of **polymorphism** enables different types of objects (in the same hierarchy) to use their own realization of the same type of action.

Virtual functions enable derived classes to define their own versions of the methods of the base class in such a way that the choice between the different versions will be made *at runtime*, according to the actual type of the object that must execute the function, *provided that it is a dynamic object* (***pointer or address***).

VI.2 Virtual functions

✓ *What is a virtual function?*

A virtual function is *declared as such in the parent class*, using the keyword `virtual` in front of the function declaration.

A *function is virtual* if it is declared virtual, or if there exists a virtual function with the same prototype in one of the base classes. It is not necessary to redefine a virtual function in all the derived classes.

A class that declares or inherits a virtual function is called a *polymorphic class*.

✓ *What is the effect of declaring a virtual function?*

Upon message passing to a pointer to an object, if the keyword `virtual` has *not* been used when declaring the corresponding method, the implementation to be

used is determined *during compilation*. Only the declared type for the object is relevant.

Example :

```
    ...
    parent_class *parent;
    derived_class *derived;
    ...
    parent = derived;
    parent->print();   // calls the function "print" of the
                       // parent class (even if "print" is
                       // redefined in derived_class)
```

But if `print` is declared `virtual`, late binding enables to take into account the ***actual type*** of the object pointed to by `parent` (not its declared type).

Exercise 14.

Assume now that classe *counter* is defined as follows:

```
class counter {
private :
  unsigned int value;
public :
  counter(unsigned int v=0) {
    value = v;
  }
  virtual unsigned int increment(){
    cout << "--> counter::increment\n";
    if (value<65535) return ++value;
    else return value;
  }
  unsigned int operator()() {
    return value;
  }
};
```

What will now be the outcome of the execution of the same `main`? (i.e., what will be printed?)

```
int main() {
  counter c0;
  range_limited_counter c1(1,10);

  c0 = c1;
  cout << "c0 (after assignment of c1): ";
  c0.increment();

  counter *pc2 = new counter(2);
  range_limited_counter *pc3 = new range_limited_counter(3,20);
```

```
  pc2 = pc3;
  cout << "pc2 (after assignment of pc3): ";
  pc2->increment();

  pc2 = new counter(2);
  pc3 = (range_limited_counter *)pc2;
  cout << "pc3 (after assignment of pc2): ";
  pc3->increment();
  return 0;
}
```

✓ We have introduced and discussed the concept of polymorphism using assignments, but *virtual functions* are particularly useful in the context where *functions* ***take as parameter (the address of) an object*** *which is of the most general type* (i.e., class), as illustrated below:

```
class A {
  ...
  public :
    virtual int fct(...);
    ...
};

class B : public A {
  ...
  public :
    int fct(...);
    ...
};

class C : public A {
  ...
  public :
    int fct(...);
    ...
};

void fct2(A *param) {  // param is of type A*
  ...
  param->fct();
  ...
}

int main() {
  ...
  A *p_obj1;
  B *p_obj2;
  C *p_obj3;
  fct2(p_obj1);
  fct2(p_obj2);   // permitted due to compatibility
```



```
  fct2(p_obj3);  // permitted due to compatibility
  ...
}
```

The method `fct` is virtual => though the type of the parameter of `fct2` is pointer to `A`, the method to be used will be chosen ***according to the type of the actual parameter***:

```
fct2(p_obj1);   → param->fct() calls fct() of A
fct2(p_obj2);   → param->fct() calls fct() of B
fct2(p_obj3);   → param->fct() calls fct() of C
```

Example 1 : let us now consider an example in which the printing method is *virtual* (and redefined in the subclasses), and is used by an independent function `print_info`

```
class parent {
  protected :
    char version;
  public :
    parent() {
      version = 'A';
    }
    virtual void print() {
      cout << "Base class, version = " << version << endl;
    }
};

class derived1 : public parent {
  private :
    int info;
  public :
    derived1(int number) {
      info = number;
      version = 'B';
    }
    void print() {
      cout << "Class derived1, info = " << info
           << "  version = " << version << endl;
    }
};

class derived2 : public parent {
  private :
    int info;
  public :
    derived2(int number) {
      info = number;
    }
```

```
    void print() {
      cout << "Class derived2, info = " << info
           << "  version = " << version << endl;
    }
};

void print_info(parent *info_holder) {
  info_holder->print();
}
```

Though the type of `info_holder` is pointer to `parent`, the function `print_info` will call the expected function.

```
int main() {
  parent b;
  derived1 d1(3);
  derived2 d2(15);
  print_info(&b);
  print_info(&d1);
  print_info(&d2);
  return 0;
}
```

which results in:

with "virtual"	*without "virtual"*
Base class, version = A	Base class, version = A
Class derived1, info = 3 version = B	Base class, version = B
Class derived2, info = 15 version = A	Base class, version = A

Exercise 15.

Write a realistic version of this example 1, that uses an **overloading of the <<** operator.

Exercise 16.

Modify the classes *counter* and *range_limited_counter* to enable an overloading of the << operator.

Adapt the function *main* of exercise 12 as follows:

- create a *counter c0*, a *range_limited_counter c1* with initial value 1 and maximum value 10, a pointer to *counter pc2* with initial value 2, and a pointer to *range_limited_counter pc3* with initial value 3 and maximum value 20
- declare a *vector v1* of *pointers to counter* and insert in this vector: the

address of *c0*, the address of *c1*, *pc2*, and *pc3*

- go through *v1* (use a const_iterator) to print the characteristics of each element.

Example 2 : this second example illustrates the fact that *a virtual function can also be called by another (non virtual) method*.

```
#include <string>
using namespace std;
class Item {   // items in s store's stock
  private :
    int ref;
    string designation;
    int quant;
  protected :
    float price;
  public :
    Item(int r, string d, float p, int q) {
      ref=r;
      designation = d;
      price=p;
      quant=q;
    }
    void print_price_with_VAT() {
      cout << "Here is the price with VAT for this object: "
           << price_with_VAT() << " euros" << endl;
    }
    virtual float price_with_VAT() {
      return (price * 1.2);
    }
    void add(int q) {
      quant = quant + q;
    }
    ...
};

class Luxury_Item : public Item {
  public :
    Luxury_Item(int r, string d, float p, int q) :
                                          Item(r,d,p,q)
    {}
    float price_with_VAT() {   // redefined
      return (price * 1.33);
    }
    // Note that print_price_with_VAT() is not virtual, not
    // redefined
};

int main() {
  Item *a1 = new Item(124, "jewel", 35, 20);
```

```
    a1->print_price_with_VAT();
    Luxury_Item *a2 =
          new Luxury_Item(563, "lux. jewel", 1200, 10);
    a2->print_price_with_VAT();
    cout << "Now the assignment..." << endl;
    a1 = a2;
    a1->print_price_with_VAT();
    return 0;
}
```

which results in:

```
Here is the price with VAT for this object: 42 euros
Here is the price with VAT for this object: 1596 euros
Now the assignment...
Here is the price with VAT for this object: 1596 euros
```

/!\ And note that **virtual destructors** are needed when deleting an instance of a derived class, through a pointer to the base class.

VI.3 Pure virtual functions

✓ The notions described above are also useful to implement *very general classes*, from which *families of more specific subclasses* will be derived.

✓ In addition, if the base class cannot give a definition for the virtual function (because it is too abstract to do so), then the function is declared as a ***pure virtual function***, using the following syntax:

```
virtual <type> <name>(...) = 0;
```

✓ Beware: when a class contains at least one pure virtual function, it is considered as an ***abstract class***, and it is ***impossible*** *to create instances* of this class.

In a derived class C_2, a pure virtual function (inherited from a base class C_1) that is not defined in C_2 is still considered pure virtual, which means that C_2 is also an abstract class.

Note. It is possible to declare a *pointer* to an object of an abstract class (or to use this type for a parameter), but not to create the object itself.

Example : the following class `Shape` can be used to characterize a family of geometric forms:

```
class Shape {
  protected:
    Coord xorig;
    Coord yorig;
    Color co;
  public:
    Shape(Coord x, Coord y, Color c) : xorig(x),
                                       yorig(y), co(c)
    {}
    ...
    virtual void draw() = 0;
};
```

Examples of derived classes can be `Circle`, `Square`,…

The class `Shape` contains a pure virtual function `draw` (that cannot have a definition as long as it does not describe how to draw a specific geometric form). This function will have actual definitions in the derived (concrete) classes.

Exercise 17.

The goal of this exercise is to deal with vectors of heterogeneous functions of one variable (for example polynomials). To that goal, we will define a general class *Function*, and concrete classes (*Poly0*, *Poly1*,…) which inherit from it.

-1- Define a class *Function* that has one attribute *name* which is a string; it identifies the type of function (e.g. "cosine", "polynomial of degree 1",…). This class also contains the following methods: *eval* that computes the value of the function at some point x_0, *derivative* that returns a pointer to a new *Function* representing the derivative, and *print* that enables to display the characteristics of a class instance. Design this method such that the class can also propose an overloading of the << operator.

-2- From this general class, derive specific subclasses to represent polynomials: class *Poly0* represents $p(x)=k0$, class *Poly1* represents $p(x)=k0 + k1.x$, and class *Poly2* represents $p(x)=k0 + k1.x + k2.x^2$ (Note. consider that *Poly0* includes the null polynomial).

-3- Define a *main* function in which you declare a vector of pointers to *Function* and you insert the following functions in this vector : $p(x)=6$, $p(x)=2 + 5.x$ and $p(x)=10 + 3.x + 2.x^2$

Then you iterate on this vector to: display the characteristics of each polynomial, and evaluate its derivative at point 1. You should get a result of the form:

```
$ ./exercise17
++ Content of the vector ++
This function is a Polynomial of degree 0
p(x) = 6
The value of its derivative at point x=1 is 0
This function is a Polynomial of degree 1
p(x) = 2 + 5.x
The value of its derivative at point x=1 is 5
This function is a Polynomial of degree 2
p(x) = 10 + 3.x + 2.x^2
The value of its derivative at point x=1 is 7
```