3. Defining and using classes

As it was said in the previous chapter, the notion of ***class*** represents a fundamental element for the data abstraction paradigm, and it is imported into the object oriented programming paradigm as a basic element.

The class can be seen as an extension of the C language structure, being the syntactic way for defining new data types. There are important differences between `struct`, `union`, and `class`. These diffrences are related, in general, to the rights of accessing the members, and to the possibility of clas derivation.

A ***class*** represents, conceptually speaking, the common properties of a collection of related objects, which justify the choice of the name `class` for this construction. The objects which belong to a class are called ***instances*** of the class or ***objects***. So, a class can be seen as a data type represented by the set of all its instances.

The compiler treats an object as any variable: a data storage zone, whose memory address is unique. In this zone there are stored the current values of the data members corresponding to the respective object, but in addition the compiler allows the member functions of the class to operate over these values.

A program that uses object oriented programming paradigm involves both the definition of the used classes, and also the declaration and the utilization of the instance of those classes that evolves by passing messages between them.

The definition of a class has two distinct parts: the ***class declaration*** and the ***class implementation***. Usually, the declaration of a class is separated by the implementation, and it is described in a header file.

The declaration part of a class must specify the the ***class name*** (and the classes from where the class is derived, if necessary) and also the ***components*** or ***members*** of the respective class. Unlike the C language, in the C++ language the components of a class (or structure or union) can be data (***data type members***), or functions (***function type members***).

Example 3.1. The structure of a simple program which uses a *stack* class defining a stack of characters can be the following:

```
// stack.h file- class declaration
class stack {
  int dim ;                 //member data
  char *buff ;              //member data
public :
  stack(int) ;              // constructor function
  ~stack() ;                // destructor function
  void push(char) ;         // member function
  char& pop() ;             // member function

} ;
```

```
// stack.cpp file – class implementation
stack::stack(int n)
{
  // constructor code
}

stack::~stack(int n)
{
  // destructor code
}

void stack::push(char c)
{
  // push function code
}

char& stack::pop()
{
  // pop function code
}

// main.cpp file – class utilization
#include <iostring>
using namespace std ;

int main() {
  //the objects declaration
  stack st1(100), st2(50) ;
  //stack objects utilization
  st1.push('a') ;
  // ...
}
```

The using of a class in an object-oriented application involves to create a set of objects, and to pass some messages between these objects. The passing mechanism and message receiving to/from objects is represented by the call of the member functions.

3.1 Class declaration

The syntax of the class declaration:

<class declaration> ::= <class header> [<member declaration>] ;
<class header> ::= <class specifier> <class name>
[: <base class> {, < base class>}]*
<member declaration> ::= '{' {<specific member>} '}'*
<class specifier> ::= struct | class
<base class> ::= [<class modifier access> :]
<class name>
<specific member > ::= [<member modifier access> :]
< member declaration >

The class heading contains mandatory the class name and the class specifier, which can be `struct` or `class`. It is possible to be also `union`, but in this case the class can not be used in the inheritance mechanism.

A class can be derived from one or more classes, which called ***base classes*** for the current class. In the case when the class declared is derived from other classes, this fact must be specified by specifying their names and their access types. The default acces type for `class` is *private,* and for `struct` the access type is *public*. The notion of the access type of the base classes will be discussed later, when discussing the inheritance mechanism.

One can observe from the previous syntax that the member declaration of a class is optional; in this case it is an uncomplete class declaration. This kind of declaration is usually used when the classes are defined recursively. Of course before declaring an instance object of a class, it first must be completely declared.

Example of an incomplete declaration. Each of the classes *ClA* and *ClB* uses a pointer to the object of the other class (the *ClB* class can not use an object, because *ClA* has not been define yet).

```
struct ClA ;
struct ClB
{
  // ...
  ClA *a ;
  // ...
} ;
struct ClA
{
  // ...
  ClB *b ;
  // ...
} ;
```

The declaration of class members (data or functions) is made as in the C language (for variables or functions). There are two exceptions representing two special categories of function members, called ***constructors*** and ***destructors*** (will be detailed later).

The declaration of a class member can be preceded, optionally, by an ***access modifier*** of the respective member (that is different from the access modifier of a base class). This access modifier can be `private`, *`protected`* *or* *`public`*. The access modifier for a class member specifies the way in which the respective member can be seen outside the class. A `public` member is visible outside, a `private` member is inaccessible, and a `protected` member can be accessed only in a class derived from the respective class with the public access modifier. So, all *protected* members are inaccessible outside, not taking in consideration the *public* derived classes. The protected type it will be discussed later.

Remark. An access modifier affects the accessibility of all declared member after this in the current class, until another access modifier is encountered. If the first declared member of class does not have specified an access modifier, then, by default this is *private* for `class` and *public* for `struct`.

Example 3.2. The *polygon* class stores pointers to the polygon vertices (not the vertex coordinates):

```
struct point {
  double x, y ;
  point double x0=0, double y0=0)
      { x = x0; y = y0; }
} ;

class polygon {
  //private members
  int nr_vertices ;
  point **vertices ;
  double area, perimeter ;
  void ComputePerimeter() ;
  void AdjustArea() ;
public :
  //public members
  polygon() ;
  ~polygon() ;
  int NrVertices() const { return nr_vertices ; }
  void AddVertex (point*) ;
  point* operator[](int) ;
  double Area() const { return area; }
  double Perimeter() const { return perimeter; }
} ;
```

Public members of a class can be accessed outside of the class and they represent the interface of that class (the way the class communicates with the exterior). The private members are local to the respective class.

The scope of the members of a class is represented by the class definition. This allows defining members in different classes with the same name, which represent different members (variables or functions).

The member functions are usually only declared in a class declarations. But, as seen in the previous example (the functions *NrVertexes, Area and Perimeter*), some simple functions can be both declared and implemented inside the declaration part of a class. These functions implemented inside the declaration of a class represent ***inline*** functions. Because the *inline* functions are expanded at compilation, their part must contain simple and small statements.

The objects of a class can be declared as constants, as other variables in a C program. In this case a problem can appear when there are called member functions of a constant object that can modify other members of the respective objects. The C++ language allows for a constant object to call only ***constant member functions*** of the class. A constant member function is specified in the class declaration with the keyword `const` specified after the function header. Such a function must not modify the member data values of the class from where it belongs (this fact is verified by the compiler).

Example 3.3.:

```
class circle {
```

```
    double xc, yc ;
    double r ;
  public :
    circle(double a, double b, double c)
       { xc = a ; yc = b ; r = c ;}
    double GetXc () const { return xc ; }
    double GetYc () const { return yc ; }
    double GetR() const { return r ; }
    void Translate(double dx, double dy)
       { xc += dx ; yc += dy ; }
  } ;
```

Circle class utilization:

```
Circle c1(0, 0, 10) ;
const circle c2(8, 7, 5) ;
c1.Translate(2, 3) ; // correct
c2.Translate(2, 3) ; // incorrect
double x = c2.GetXc () ;   // correct
```

In a similar way to the constant objects, a C++ program can uses ***volatile*** objects declared with the keyword `volatile`. In this case the C++ compiler supposes that the state of a volatile object can be modified outside of the program and it does not perform some operations (for example the code optimization operation). For a volatile object only a ***volatile member function*** for that object can be called. A volatile member function can de defined similarly with the constant functions replacing the `const` keyword by `volatile`.

Example 3.4.: A class that controls a hardware device by placing appropriate values in hardware registers at known absolute addresses.

```
// file devregs.h
// Declare the device registers
struct devregs{
  // control-status-register
  unsigned short volatile csr;
  // data
  unsigned short const volatile data;
  // Busy-wait function to read a byte from device
  unsigned int read_dev() volatile;
  // constructor
  devregs();
};

// file devregs.cpp
// bit patterns in the control-status-register
#define ERROR   0x1
#define READY   0x2
#define RESET   0x4

devregs::read_dev() {
  csr = 0;
  data = 0;
}

unsigned int devregs::read_dev() volatile {
  while((csr & (READY | ERROR)) == 0)
```

```
    ; // NULL - wait till done
  if(csr & ERROR){
    csr = RESET;
    return 0xffff;
  }
  return(data & 0xff);
}

// file main.cpp
#include devregs.h
void process(void) {
  volatile devregs dvp;
  unsigned int ret;
  ret = dvp->read_dev();   // OK
}
```

Access functions represent a group of member functions very used in C++ programs. These are functions, usually defined as *inline*, which allow to read or to modify the value of the private data members of the classes, where the user does not have direct access. The functions reading values are usually called *accessors*, while the functions writing values are called *modifiers.*

There are not predefined rules for naming these functions, but usually the accessors are prefixed by *Get*, while modifiers are prefixed by *Set*. For example, the functions *GetXc*, *GetYc* and *GetR* are accessors. A modifier can be defines as:

```
void SetXc(double x) { xc = x; }
```

Another way used is that of writing overloaded functions, for accessors and for modifiers. For example for member data *xc* of *Circle* class, the following access functions can be defined:

```
void Xc(double x) { xc = x; }
double Xc() const { return xc; }
```

Remark. It is not recommended that accessors to return references, nor not constant pointers at the private data of the classes (in this case they allow the direct access to the private data members).

3.2 Class implementation. The resolution operator

For a complete class definition all member functions from the respective class declarations that are not *inline* must be implemented. Usually the implementation of *non-inline* functions is made in a distinct source file.

The C++ language has a new operator called ***resolution operator*** (denoted ::), in order to be able to specify in the case of every member function the class where it belongs. The complete specification of a function name, using the resolution operator is the following:

<class name> : : <function name>

For example, the functions *AddVertex*, *ComputePerimeter* and *Adju*stArea from the *polygo*n class can be defined as follows:

```
void polygon::AddVertex(point* p) {
  point **v = new point *[nr_vertices+1] ;
  for(int i=0 ; i< nr_vertices ; i++)
    v[i] = vertices[i] ;
  v[nr_vertices++] = p ;
  delete[] vertices ; //free the memory for vertices
  vertices = v;
  Compute Perimeter();
  AdjustArea();
}

void polygon::AdjustArea(void)  {
  if (nr_vertexes > 2)
    area = area+( vertices[0]->x * vertices[nr_vertices-2]->x +
      vertexes[nr_vertices-2]->y * verteics[nr_vertices-1]->x +
      vertexes[nr_vertices-1]->y * vertices[0]->x -
      vertexes[0]->x * vertexes[nr_vertices-2]->y -
      vertexes[nr_vertices-2]->x * vertices[nr_vertices-1]->y -
      vertexes[nr_vertices-1]->x * vertices[0]->y ) / 2;
}

void polygon::ComputePerimeter(void) {
  double l;
  if (nr_ vertexes > 1)
    for (int i=0; i<nr_ vertices -1; i++) {
      l = sqrt((vertices[i]->x –vertices[i+1]->x) *
          (vertices[i]->x - verices[i+1]->x) +
          (vertices[i]->y - verices[i+1]->y) *
          (vertices[i]->y - vertices[i+1]->y));
      perimeter += l;
    }
    l = sqrt((vertices[0]->x - vertices[nr_vertices-1]->x) *
        (vertices[0]->x - vertices[nr_vertices-1]->x) +
        (vertices[0]->y - vertices[nr_vertices-1]->y) *
        (vertices[0]->y - vertices[nr_vertices-1]->y));
    perimeter += l;
}
```

The resolution operator can be used also in other cases. A usual utilization is the reference of a name which is hidden in a block. For example, a variable defined in a file outside any function is visible in all functions from the respective file, excepting the blocks where it is redefined:

```
int k ;
f1()
{
  // k is visible in this block
}

f2()
{
  // k defined at the file level it is not visible in f2
  int k = 0 ;
  k = k+2 ;     // using k at the block level
```

```
  ::k = ::k+2 ; // using k at the file level
}
```

The resolution operator can be used in this case as a unary operator which prefixes a name; in this case it refers the most outside appearance of the respective name, declared at the file level. In the previous example, the instruction:

```
k = k+2 ;
```

refers to the variable defined in the function *f2*, while:

```
::k = ::k+2 ;
```

refers the variable defined at the file level.

The class ***constructors*** and the ***destructors*** are member functions of the respective class. In the case when they are not defined as inline functions, they must be defined in the implementation file of the class. For example, the constructor for *polygon* class can be defined as follows:

```
polygon::polygon()    // constructor
{
  vertices = 0 ;
  nr_vertices = 0 ;
  area = perimeter = 0;
}

polygon::~polygon()   // destructor
{
  delete[] vertices ;
}
```

As seen before, constructors and destructors are special functions that do not have any returned value (not even `void`). Moreover, the destructors can not have arguments.

Remark. The constructor of the class *polygon* does not allocate memory for polygon vertices. Adding of a vertex can be realized by using the function *AddVertex*, but the memory allocation must be realized outside of this function.

Another special category of member functions are the ***operator***s. The C++ language allows overloading the common operators of the language in order to be able to define other operations. The specification of an operator as a function is done with the help of the keyword `operator` used as prefix of the respective operator. In the example of *polygon* class it is overloaded the indexing operator:

```
point* polygon::operator[](int k)     {
  if (k < 0) {
    cout << "\n Negativ index" ;
    return 0 ;
  }
  return vertices[k] ;
}
```

3.3 Using classes

As seen before, ***using of a class*** means the creation of some instance objects of these classes and the communication with the respective objects with the help of messages, that is calling their member functions.

The instance objects resemble with variables, e.g. they have a reserved memory zone for storage of member data. All the observation referring to storage zones for variables are valid also for objects, so the memory allocation for objects can be made in the *static data* part, or in the *stack* part, or in the *heap* part of the program memory.

The creation of an object involves two distinct operations:

- The allocation of a memory zone having an appropriate dimension;
- The call of a constructor function of the class where the object belongs to, in order to initialize the member data with initial values.

The dimension of the memory zone allocated for an instance object is, in general, given by the sum of the dimensions of the data members, but this dimension depends on implementation. There are situations when the memory zone dimension of an object is greater than this sum, especially in the case of polymorphism, or in the case of some classes which do not contain only member functions.

For example, for a Visual C++ compiler, the sequence:

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

struct A {
  int n;
  A(int k) { n = k ;}
  int N() { return n; }
};

struct B {
  void Print() { cout << "B"; }
};

int main() {
  A a;
  B b;
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
  return 0;
}
```

displays the values 4 and 1, because the `int` values are represented on 4 octets. The 1 value appears because the compiler does not allow the existence of an object with zero dimensions.

The call of an object member function is strictly related with the respective object, by passing of a hidden parameter, which refers the memory address of the respective.

This aspect will be discussed taking in consideration an example. For an application which works with polygons to be able to use more polygons stored into a doubled linked list, at the *polygon* class must be added 2 new members having the pointer type, which respectively indicates to the precedent polygon and to the next one. The *polygon* class can be rewritten (here it is not a derived class, but a new distinct class):

```
class p_polygon {
  int nr_vertices ;
  point **vertices ;
  double area, perimeter ;
  p_polygon *succ, *pred ;
  void ComputePerimeter() ;
  void AdjustArea() ;
public :
  p_polygon()
  {
    vertices = 0 ;
    nr_vertices = 0 ;
    area = perimeter = 0 ;
    succ = pred = 0;
  }
  ~p_polygon() ;
  int NrVertices() const { return nr_vertices ; }
  void AddVertices (point*) ;
  point* operator[](int) ;
  double Area() const { return area; }
  double Perimeter() const { return perimeter; }
  p_polygon* Pred() const { return pred ; }
  p_polygon* Succ() const { return succ ; }
  void AddPolygon(p_polygon*) ;
} ;
```

3.3.1 The keyword *this*

The implementation of the functions of the previous class p*_polygon* is the same as in the case of the class *polygon*, with the exception of the new added one. The *AddPolygon* function adds a new polygon in the list, as successor of the current polygon:

```
void p_polygon::AddPolygon(p_polygon* p) {
  p->succ = succ ;
  p->pred = this ;
  succ->pred = p;
  succ = p ;
}
```

It is observed the utilization of a new keyword, `this`. This is called the *self pointer* and it points always on the current object. `This` can be seen as an invisible parameter in the *AddPolygo*n function declared as:

```
p_polygon* this
```

Remark. In the case of a class *X*, the parameter:

X* *this*

is passed in all the non-static member functions of the *X* class.

A constructor for an object is implicitly called at the creation of an instance object of a class after the memory allocation. This constructor will have `this` as a hidden parameter initialized with the block address associated to the object.

Let us suppose that in the current application are defined two objects:

```
p_polygon* p = new p_polygon ;
// ...
p_polygon d ;
// ...
```

In both cases it is called the class constructor *p_polygon*: in the first case it is default called by the `new` operator, and the memory is allocated in the *heap* zone of the program, while in the second case the constructor is also default called by the compiler, but the memory for the object is allocated in the data zone.

In both cases, the created objects will contain a copy of all member data of the *p_polygon* class, and some of the members (*nr_vertices*, *vertices*, *area*, *perimeter*, *succ*, and *pred*, for example) will be initialized by the constructors with certain values (0 in our case). In addition all *'this'* parameters of the member functions for every object will be initialized with the memory block address associated to object (in the first case, for example, with the value of *p* pointer).

Let us suppose that it is created a new object:

```
p_polygon* p4 = new p_polygon ;
// ...
```

The adding of *p4* in the polygons list after the *p* polygon can be describes as follows:

```
p->AddPoligon(p4) ;
```

Now can be observed the utility of `this` pointer, because each new created object of the *p_pol*ygon class will have another memory address, which is necessary when adding in the polygon list.

The using of the hidden parameter `this` is not absolutely necessary, only in the case when an explicitly reference at the memory address of the current object is made. For example, the function *AddPolygon* can be also written as follows:

```
void p_polygon::AddPolygon(p_polygon* p) {
  p->succ = this->succ ;
  p->pred = this ;
  this->succ->pred = p;
  this->succ = p ;
}
```

Remark. When pointer to an object is used, the class constructor it is not called if the `new` operator is not called. For example, the declaration:

```
p_poligon* pp;
```

does not create any object of the *p_pol*ygon class.

3.3.2 Static members of a class

As seen before, every instance object of a class has usually a copy of the member data of the class to which it belongs. For this reason any modification of the value of an object member is local to the respective instance and it is not visible in other instances of the same class.

The C++ language allows, in addition, the possibility of defining members having values that can be used in common by all other class instances. These members are called ***static members*** and they are declared with by using the keyword `static`.

Example 3.5. Let us consider a class *Experiment*, which allows the description of the observations on a physic measure. Each class object stores a measured value of the physic measure. The *Experiment* class must determine the observations number at a certain time, and also the average of the observed values. Two static member data are used (*n* and *s*), which store the number of created objects until the current time and their sum of values, and also two static member functions (*N* and *Med*) for retrieving the number observations and the average of these values.

```
// experiment.h file
class Experiment {
  double x;
  static int n;
  static double s;
public:
  Experiment(double);
  double X() const { return x; }
  static double Med() { return s/n; }
  static int N() { return n; }
};
```

Member data having static type are common for all class objects and they have allocated a memory zone that is different from the zone of non-static data. In this way it can be realized the simple and efficient communication between different objects belonging to the same class.

The effective definition (memory allocation and initialization with values) of the static member data must be realized outside of the class declaration and in a single place in the program. Usually the definition of the static member date is realized in the file which contains the implementation of the class, avoiding in this way multiple definitions.

For the previous example, in the implementation file of the experiment class must be added the following definitions:

```
int Experiment::n = 0 ;
double Experiment::s = 0 ;
```

as in the next example:

```
// experiment.cpp file
int Experiment::n = 0;
double Experiment::s = 0;

Experiment::Experiment(double v) {
  x = v;
```

```
  n++;
  s += v;
}
```

The constructor of the *Experiment* class must do two supplementary actions for every new created object: incrementation of the total number of the class objects, and also adding of the object current value to the sum of all values of the class objects.

The static data can have any type of access (`public`, `protected`, `private`), as any class member. The conclusion that these static data members can be used anywhere in the program is false: being static, the compiler does not allow another definition for them and the value modification outside of the class where they have been declared also it is not allowed.

The using of static member data lead to a better structuring of information in a program, because these values are global only for class objects where they have been declared.

In the previous example, *N* and *Med* are two ***static member functions*** of the *Experimen*t class. As static member data, the static member functions of a class are unique for all respective class instances. Their using is necessary when certain classes have static member data.

Static member data can not use the hidden parameter `this`. From this aspect results another conclusion: *the static member data can have access only at the static members of the respective class* (data or functions). In the previous example, the *Med* function can not modify the *x* member value and it can not call other member functions of the class (*X* for example).

One way that a static member function can have access at the non-static member of the class to which it belongs, is by passing as parameter in the static function of an object of the respective class.

Example 3.6. The class *Folder* stores the path for a current folder and a predefined path, unique for all class objects. The static function *preset* allows setting the current path for a folder which is passed as parameter.

```
class Folder {
public:
  static void setpath(char const *newpath);
  static void preset(Folder &dir, char const *path);
private:
  string Currentpath;
  static char path[];
};

char Folder::path[200] = "C:\\";

void Folder::setpath(char const * newpath) {
  strcpy(path, newpath);
}

void Folder::preset(Folder &dir, char const * newpath)
{
  dir.Currentpath = newpath;
}
```

```
int main() {
   Folder dir;
   Folder::setpath("D:\\");
   dir.setpath("D:\\");
   Folder::preset(dir, "D:\\OOP");
   dir.preset(dir, "D:\\OOP");
   return 0;
}
```

Another particularity of these functions is that they can be called also directly, even if they are declared *private*, without the help of an object from from the class to which it belongs. For example, a file using the *experiment* class could be the following (it is considered the sequence of the following values: 0.5, 1.5. 2.5, … , 9.5):

```
//main.cpp file
int main() {
  for (int i=0; i<10; i++)
    Experiment e(i+0.5);
  int n = Experiment::N();
  double m = Experiment::Med();
  cout << "n = " << n << endl << "Med = " << m << endl;
  return 0;
}
```

It is observed that the functions *N* and *Med* have been called without any *experiment* class object.

Remarks.

1. The non-static member functions can refer static members of the respective class (for example, the case of the constructor of *Experiment* class).
2. A static member function which is private can not be called by means of a class object.
3. If a member function is declared *static* in a class, but it not defined as *inline*, the effective definition of this function does not contain the word `static`. For example:

```
class A
{
  // ...
  static int x ;
  static void SetX(int) ;
  // ...
} ;

void A::SetX(int k)
{
  x = k ;
}
```