Object Oriented Programming:

15. Classes (I)

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword `class`, with the following format:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
  } object_names;
```

Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords: `private`, `public` or `protected`. These specifiers modify the access rights that the members following them acquire:

- `private` members of a class are accessible only from within other members of the same class or from their *friends*.
- `protected` members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- Finally, `public` members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the `class` keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access.

For example:

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
  } rect;
```

Declares a class (i.e., a type) called `CRectangle` and an object (i.e., a variable) of this class called `rect`. This class contains four members: two data members of type `int` (member `x` and member `y`) with private access (because private is the default access level) and two member functions with public access: `set_values()` and `area()`, of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, `CRectangle` was the class name (i.e., the type), whereas rect was an object of type `CRectangle`. It is the same relationship `int` and `a` have in the following declaration:

```
int a;
```

where `int` is the type name (the class) and `a` is the variable name (the object).

After the previous declarations of `CRectangle` and `rect`, we can refer within the body of the program to any of the public members of the object `rect` as if they were normal functions or normal variables, just by putting the object's name followed by a dot (`.`) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are `x` and `y`, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:

```
// classes example
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};
```

```
void CRectangle::set_values (int
a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " <<
rect.area();
  return 0;
}
```

```
area: 12
```

The most important new thing in this code is the operator of scope (`::`, two colons) included in the definition of `set_values()`. It is used to define a member of a class from outside the class declaration itself.

You may notice that the definition of the member function `area()` has been included directly within the definition of the `CRectangle` class given its extreme simplicity, whereas `set_values()` has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (`::`) to specify that we are defining a function that is a member of the class `CRectangle` and not a regular global function.

The scope operator (`::`) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function `set_values()` of the previous code, we have been able to use the variables `x` and `y`, which are private members of class `CRectangle`, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class and to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members `x` and `y` have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function `set_values()`. Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects

it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. For example, following with the previous example of class `CRectangle`, we could have declared the object `rectb` in addition to the object `rect`:

```
// example: one class, two
objects
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int
a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " <<
rect.area() << endl;
  cout << "rectb area: " <<
rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

In this concrete case, the class (type of the objects) to which we are talking about is `CRectangle`, of which there are two instances or objects: `rect` and `rectb`. Each one of them has its own member variables and member functions.

Notice that the call to `rect.area()` does not give the same result as the call to `rectb.area()`. This is because each object of class CRectangle has its own variables `x` and `y`, as they, in some way, have also their own function members `set_value()` and `area()` that each uses its object's own variables to operate.

That is the basic concept of *object-oriented programming*: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions

embedded as members. Notice that we have not had to give any parameters in any of the calls to `rect.area` or `rectb.area`. Those member functions directly used the data members of their respective objects `rect` and `rectb`.

Constructors and destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function `area()` before having called function `set_values()`? Probably we would have gotten an undetermined result since the members `x` and `y` would have never been assigned a value.

In order to avoid that, a class can include a special function called `constructor`, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even `void`.

We are going to implement `CRectangle` including a constructor:

```
// example: class constructor
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area () {return
(width*height);}
};

CRectangle::CRectangle (int a,
int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " <<
rect.area() << endl;
  cout << "rectb area: " <<
rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

As you can see, the result of this example is identical to

the previous one. But now we have removed the member function `set_values()`, and have included instead a constructor that performs a similar action: it initializes the values of `x` and `y` with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even `void`.

The *destructor* fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
// example on constructors and
destructors
#include <iostream>
using namespace std;

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area () {return (*width
* *height);}
};

CRectangle::CRectangle (int a,
int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}
```

```
CRectangle::~CRectangle () {
  delete width;
  delete height;
}

int main () {
  CRectangle rect (3,4), rectb
(5,6);
  cout << "rect area: " <<
rect.area() << endl;
  cout << "rectb area: " <<
rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

```
// overloading class
constructors
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return
(width*height);}
};

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a,
int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " <<
rect.area() << endl;
  cout << "rectb area: " <<
```

```
rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 25
```

In this case, `rectb` was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both `width` and `height` with a value of 5.

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses `()`:

```
CRectangle rectb;   // right
CRectangle rectb(); // wrong!
```

Default constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
  };
```

The compiler assumes that `CExample` has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
CExample ex;
```

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

```
class CExample {
  public:
    int a,b,c;
    CExample (int n, int m) { a=n; b=m; };
    void multiply () { c=a*b; };
  };
```

Here we have declared a constructor that takes two parameters of type int. Therefore the following object declaration would be correct:

```
CExample ex (2,3);
```

But,

```
CExample ex;
```

Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.

But the compiler not only creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared if you do not declare your own. These are the *copy constructor*, the *copy assignment operator*, and the default destructor.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object. For `CExample`, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (const CExample& rv) {
  a=rv.a;  b=rv.b;  c=rv.c;
  }
```

Therefore, the two following object declarations would be correct:

```
CExample ex (2,3);
CExample ex2 (ex);   // copy constructor (data copied from ex)
```

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

```
CRectangle * prect;
```

is a pointer to an object of class `CRectangle`.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (`->`) of indirection. Here is an example with some possible combinations:

```
// pointer to classes example
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
```

```
    int area (void) {return
(width * height);}
};

void CRectangle::set_values (int
a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle a, *b, *c;
  CRectangle * d = new
CRectangle[2];
  b= new CRectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area()
<< endl;
  cout << "*b area: " << b-
>area() << endl;
  cout << "*c area: " << c-
>area() << endl;
  cout << "d[0] area: " <<
d[0].area() << endl;
  cout << "d[1] area: " <<
d[1].area() << endl;
  delete[] d;
  delete b;
  return 0;
}
```

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

Next you have a summary on how can you read some pointer and class operators (*, &, ., ->, [ ]) that appear in the previous example:

expression	can be read as
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
x[0]	first object pointed by x
x[1]	second object pointed by x
x[n]	(n+1)th object pointed by x

Be sure that you understand the logic under all of these expressions before proceeding with the next sections. If you have doubts, read again this section and/or

data structures.

Classes defined with struct and union

Classes can be defined not only with keyword `class`, but also with keywords `struct` and `union`.

The concepts of class and data structure are so similar that both keywords have in C++ the exact same functionality except that the members of classes declared with keyword `struct` have public access by default, instead of private access, as classes declared with keyword `class` have. That is the only difference. For all other purposes both keywords are equivalent.

The concept under unions is different from than of `struct` and `class`, since unions only store one data member at a time, but they are also classes and thus can also hold function members. The default access in union classes is public.