Classes

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.

http://cplusplus.com/doc/tutorial/classes/

- 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.
- 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.

```
1.  // classes example
2.  #include <iostream>
3.  using namespace std;
4.
5.  class Crectangle
6.  {
7.      int x, y;
8.    public:
9.      void set_values (int,int);
10.     int area () {return (x*y);}
11. };
12.
13. void CRectangle::set_values (int a, int b)
14. {
15.   x = a;
16.   y = b;
17. }
18.
19. int main ()
20. {
21.   CRectangle rect;
22.   rect.set_values (3,4);
23.   cout << "area: " << rect.area();
24.   return 0;
25. }
```

area: 12

```
// 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;
}
```

```
1.  // example: one class, two objects
2.  #include <iostream>
3.  using namespace std;
4.
5.  class CRectangle
6.  {
7.      int x, y;
8.    public:
9.      void set_values (int,int);
10.     int area () {return (x*y);}
11. };
12.
13. void CRectangle::set_values (int a, int b)
14. {
15.   x = a;
16.   y = b;
17. }
18.
19. int main ()
20. {
21.   CRectangle rect, rectb;
22.   rect.set_values (3,4);
23.   rectb.set_values (5,6);
24.   cout << "rect area: " << rect.area() << endl;
25.   cout << "rectb area: " << rectb.area() << endl;
26.   return 0;
27. }
```

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

```
// example: one class, two named objects
// Direct instance
#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;
}
```

```
// example: one class, two unnamed objects
//Indirect instance
#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 = new CRectangle;
   CRectangle* rectb = new CRectangle;
   rect->set_values (3,4);
   rectb->set_values (5,6);
   cout << "rect area: " << rect->area() << endl;
   cout << "rectb area: " << rectb->area() << endl;
   return 0;
}
```

Constructors and destructors

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. 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.

Neither the constructor prototype declaration (within the class) nor the latter constructor definition can include a return value; not even void

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments

```
1.  // example: class constructor
2.  #include <iostream>
3.  using namespace std;
4.
5.  class CRectangle
6.  {
7.      int width, height;
8.    public:
9.      CRectangle (int,int);
10.     int area () {return (width*height);}
11. };
12.
13. CRectangle::CRectangle (int a, int b)
14. {
15.   width = a;
16.   height = b;
17. }
18.
19. int main ()
20. {
21.   CRectangle rect (3,4);
22.   CRectangle rectb (5,6);
23.   cout << "rect area: " << rect.area() << endl;
24.   cout << "rectb area: " << rectb.area() << endl;
25.   return 0;
26. }
```

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

```
// 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;
}
```

A 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 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.

```
1.  // example on constructors and destructors
2.  #include <iostream>
3.  using namespace std;
4.
5.  class CRectangle
6.  {
7.      int *width, *height;
8.    public:
9.      CRectangle (int,int);
10.     ~CRectangle ();
11.     int area () {return (*width * *height);}
12.};
13.
14.CRectangle::CRectangle (int a, int b)
15.{
16.  width = new int;
17.  height = new int;
18.  *width = a;
19.  *height = b;
20.}
21.
22.CRectangle::~CRectangle ()
23.{
24.  delete width;
25.  delete height;
26.}
27.
28.int main ()
29.{
30.  CRectangle rect (3,4), rectb (5,6);
31.  cout << "rect area: " << rect.area() << endl;
32.  cout << "rectb area: " << rectb.area() << endl;
33.  return 0;
34.}
```

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

```
// 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;
}
```

Overloading Constructors

A constructor can also be overloaded with more than one function that have the same name but different types or number of parameters.

For overloaded functions the compiler will call the one whose parameters match the arguments used in the function call.

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:

```
1. // overloading class constructors
2. #include <iostream>
3. using namespace std;
4.
5. class CRectangle {
6.     int width, height;
7.   public:
8.     CRectangle ();
9.     CRectangle (int,int);
10.    int area (void) {return (width*height);}
11.};
12.
13.CRectangle::CRectangle () {
14.  width = 5;
15.  height = 5;
16.}
17.
18.CRectangle::CRectangle (int a, int b) {
19.  width = a;
20.  height = b;
21.}
22.
23.int main () {
24.  CRectangle rect (3,4);
25.  CRectangle rectb;
26.  cout << "rect area: " << rect.area() << endl;
27.  cout << "rectb area: " << rectb.area() << endl;
28.  return 0;
29.}
```

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

```
// 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;
}
```

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

```
1.  // pointer to classes example
2.  #include <iostream>
3.  using namespace std;
4.
5.  class CRectangle {
6.      int width, height;
7.    public:
8.      void set_values (int, int);
9.      int area (void) {return (width * height);}
10. };
11.
12. void CRectangle::set_values (int a, int b) {
13.   width = a;
14.   height = b;
15. }
16.
17. int main () {
18.   CRectangle a, *b, *c;
19.   CRectangle * d = new CRectangle[2];
20.   b= new CRectangle;
21.   c= &a;
22.   a.set_values (1,2);
23.   b->set_values (3,4);
24.   d->set_values (5,6);
25.   d[1].set_values (7,8);
26.   cout << "a area: " << a.area() << endl;
27.   cout << "*b area: " << b->area() << endl;
28.   cout << "*c area: " << c->area() << endl;
29.   cout << "d[0] area: " << d[0].area() << endl;
30.   cout << "d[1] area: " << d[1].area() << endl;
31.   delete[] d;
32.   delete b;
33.   return 0;
34. }
```

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

```
// 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;
}
```

Overloading Operators

C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example:

```
int a, b, c;
a = b + c;
```

This is obviously valid code in C++, since the different variables of the addition are all fundamental types. Nevertheless, it is not so obvious that we could perform an operation similar to the following one:

```
struct {
  string product;
  float price;
} a, b, c;
a = b + c;
```

In fact, this will cause a compilation error, since we have not defined the behavior our class should have with addition operations.

However, thanks to the C++ feature to overload operators, we can design classes able to perform operations using standard operators. Here is a list of all the operators that can be overloaded:

Overloadable operators

```
+     -     *     /     =     <     >     +=    -=    *=    /=    <<    >>
<<=   >>=   ==    !=    <=    >=    ++    --    %     &     ^     !     |
~     &=    ^=    |=    &&    ||    %=    []    ()    ,     ->*   ->
new
delete    new[]     delete[]
```

To overload an operator in order to use it with classes we declare operator functions, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload. The format is:

type operator sign (parameters) { /*...*/ }

```
// vectors: overloading operators example
#include <iostream>
using namespace std;
class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};
CVector::CVector (int a, int b) {
  x = a;
  y = b;
}
CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}
int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  c = a.operator+(b);
  return 0;
}
```

The keyword this

The keyword this represents a pointer to the object whose member function is being executed.
It is a pointer to the object itself.

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

class CDummy {
  public:
    int isitme (CDummy& param);
};

int CDummy::isitme (CDummy& param)
{
  if (&param == this) return true;
  else return false;
}

int main () {
  CDummy a;
  CDummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b";
  return 0;
}
```

Static members

```
#include <iostream>
//// Persistence – only 1 copy through multiple instances
using namespace std;

class myclass {
static int i;
public:
void setInt(int n) {
i = n;
}
int getInt() {
return i;
}
};

int myclass::i; // Definition of myclass::i. i is still private to myclass.

int main()
{
myclass object1, object2;
object1.setInt(10);
cout << "object1.i: " << object1.getInt() << '\n'; // displays 10
cout << "object2.i: " << object2.getInt() << '\n'; // also displays 10
return 0;
}
```