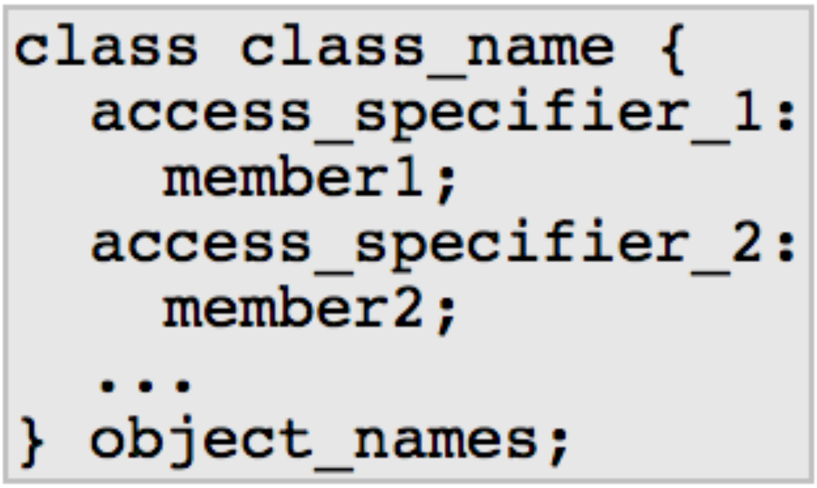
CLASSES (I)
They can contain data members, but they can also contain functions as members.

instantiation = örnekleme
paradigm = örnek, yaklaşım
respective = her biri kendisinin olan, ayrı ayrı
resorting = birbirinden ayırma
hence = dolayısıyla, sebebiyle



```
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***, which can either be **data or function** declarations, and optionally ***access specifiers*.**

<u>access specifiers:</u>

- **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 other members of the same class (or 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 any other *access specifier* has private access** automatically.

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

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

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

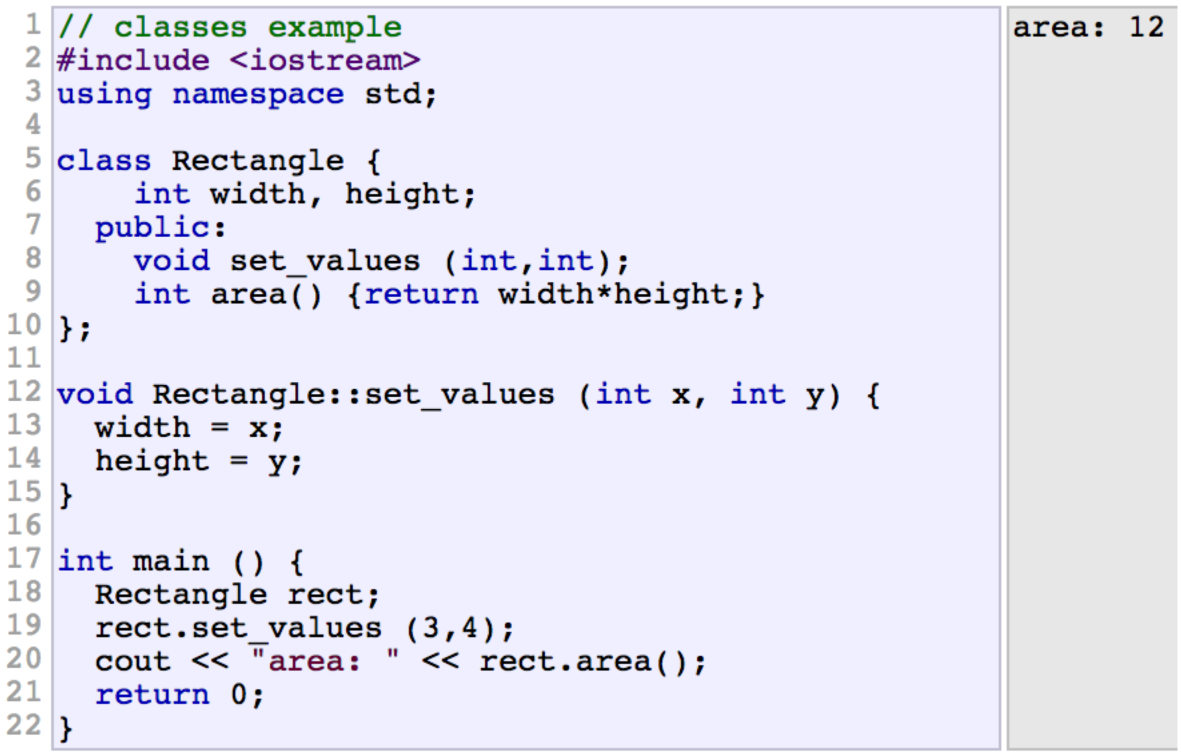
```
int a;
```

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

Any of the public members of object **rect** can be accessed as if they were normal functions or normal variables, by simply inserting a dot **(.)** between ***object name*** and ***member name***. This follows the same syntax as accessing the members of plain data structures.

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

The only members of **rect** that cannot be accessed from outside the class are **width** and **height**, since they have private access and they can only be referred to from within other members of that same class.



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

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

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

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

```
area: 12
```

The ***scope operator*** (**::**, two colons) is used in the definition of function **set_values** to define a member of a class outside the class itself.

Notice that the definition of the member function area has been included directly within the definition of class Rectangle given its extreme simplicity. Conversely, set_values it is merely declared with its prototype within the class, but its definition is outside it. In this outside definition, the operator of scope (::) is used to specify that the function being defined is a member of the class Rectangle and not a regular non-member 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.

The only difference **between**
defining a member function completely within the class definition or to just include its declaration in the function(the function is automatically considered an ***inline* member function** by the compiler)
and
define it later outside the class, it is a normal (not-inline) class member function.
*This causes no differences in behavior, but only on possible compiler optimizations.

Members width and height have private access (remember that if nothing else is specified, all members of a class defined with keyword class have private access). By declaring them private, access from outside the class is not allowed.

The most important property of a class is that it is a type, and as such, we can declare multiple objects of it.

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

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

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle 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
```

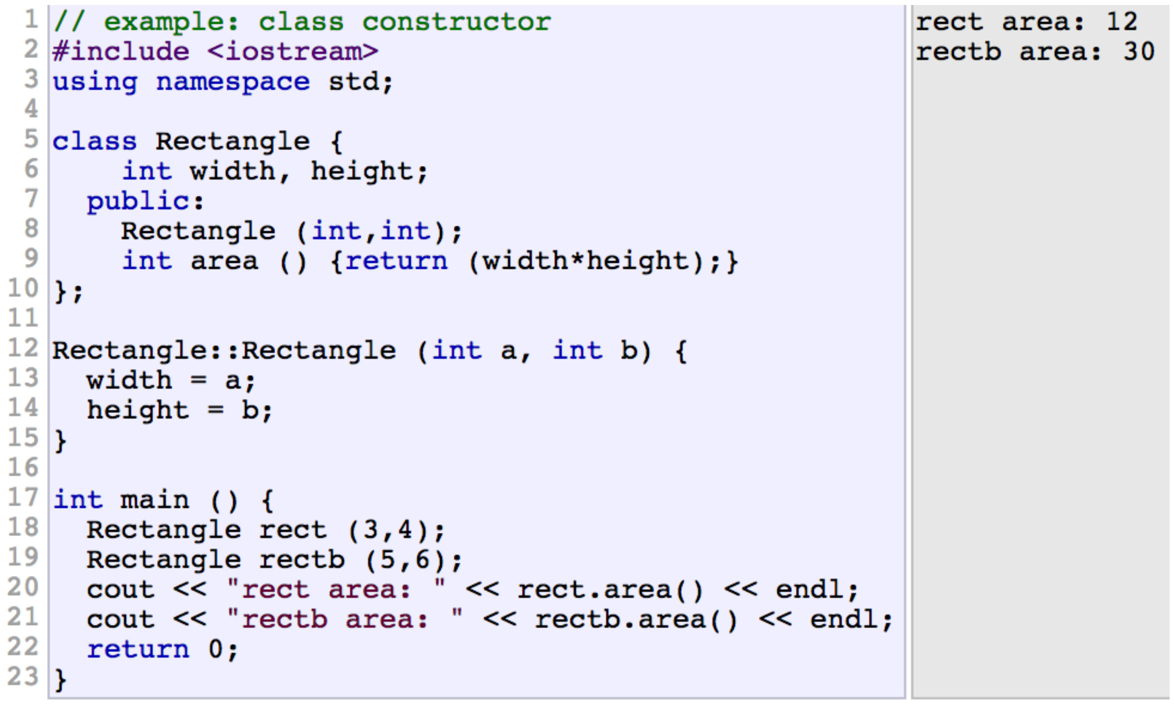
In this particular case, the class (type of the objects) is Rectangle, of which there are two instances (i.e., objects): **rect** and **rectb**.
Each one of them has its own member variables and member functions.

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 Rectangle has its own variables width and height**, as they -in some way- have also their own function members **set_value** and **area** that operate on the object's own member variables.

CONSTRUCTORS
A class can include a special function called its ***constructor***, which *is automatically called whenever a new object of this class is created*, allowing the class to initialize member variables or allocate storage.

This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void. They simply initialize the object.

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



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

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

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

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

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

The results of this example are identical to those of the previous example. But now, class Rectangle ***has no member function*** **set_values**, and has instead a constructor that performs a similar action: *it initializes the values of* ***width*** *and* ***height*** with the arguments passed to it.

OVERLOADING CONSTRUCTORS

Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments:

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

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

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

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

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

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

In the above example, two objects of class Rectangle are constructed: rect and rectb. rect is constructed with two arguments, like in the example before.

The ***default constructor*** is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments.
In the example above, the ***default constructor*** is called for **rectb**. Note how **rectb** is not even constructed with an empty set of parentheses - in fact, ***empty parentheses cannot be used to call the default constructor***:

```cpp
Rectangle rectb;   // ok, default constructor called
Rectangle rectc(); // oops, default constructor NOT called
```

This is because the empty set of parentheses would make of **rectc** a **function declaration** instead of an object declaration: It would be a function that takes no arguments and returns a value of type Rectangle.

UNIFORM INITIALIZATION
Constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument):

```
class_name object_name = initialization_value;
```

Uniform initialization, which essentially is the same as the functional form, but using braces **{ }** instead of parentheses **( )**:

```
class_name object_name { value, value, value, ... }
```

Optionally, this last syntax can include an equal sign before the braces.

```cpp
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum() << '\n';
  return 0;
}
```

```
foo's circumference: 62.8319
```

An advantage of uniform initialization over functional form is that, unlike parentheses, **braces cannot be confused with function declarations**, and thus can be used to explicitly call default constructors:

```cpp
Rectangle rectb;   // default constructor called
Rectangle rectc(); // function declaration (default constructor NOT called)
Rectangle rectd{}; // default constructor called
```

MEMBER INITIALIZATION IN CONSTRUCTORS

```cpp
class Rectangle {
    int width,height;
  public:
    Rectangle(int,int);
    int area() {return width*height;}
};
```

The constructor for this class could be defined, as usual, as:

```cpp
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

But it could also be defined using *member initialization* as:

```cpp
Rectangle::Rectangle (int x, int y) : width(x) { height=y; }
```

Or even:

```cpp
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

Note how in this last case, the constructor does nothing else than initialize its members, hence it has an **empty function body**.

For members of fundamental types, it makes no difference which of the ways above the constructor is defined, because they are not initialized by default,
but for **member objects** (those whose type is a class), if they are not initialized after the colon, they are **default-constructed.**

Default-constructing all members of a class may or may always not be convenient: in some cases, this is a waste (when the member is then reinitialized otherwise in the constructor),
but in some other cases, default-construction is not even possible (when the class does not have a default constructor). In these cases, members shall be initialized in the member initialization list.

```cpp
// member initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
  public:
    Cylinder(double r, double h) : base (r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo (10,20);

  cout << "foo's volume: " << foo.volume() << '\n';
  return 0;
}
```

```
foo's volume: 6283.19
```

In this example, class **Cylinder** has a member object whose type is another class (base's type is Circle).
Because objects of class **Circle** can only be constructed with a parameter, Cylinder's constructor needs to call base's constructor, and the only way to do this is in the ***member initializer list***.

```cpp
Cylinder::Cylinder (double r, double h) : base{r}, height{h} { }
```

http://www.cplusplus.com/reference/initializer_list/initializer_list/

POINTERS TO CLASSES
Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer.

```cpp
Rectangle * prect;
```

prect is a pointer to an object of class **Rectangle**.

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

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


int main() {
  Rectangle obj (3, 4);
  Rectangle * foo, * bar, * baz;
  foo = &obj;
  bar = new Rectangle (5, 6);
  baz = new Rectangle[2] { {2,5}, {3,6} };
  cout << "obj's area: " << obj.area() << '\n';
  cout << "*foo's area: " << foo->area() << '\n';
  cout << "*bar's area: " << bar->area() << '\n';
  cout << "baz[0]'s area:" << baz[0].area() << '\n';
  cout << "baz[1]'s area:" << baz[1].area() << '\n';
  delete bar;
  delete[] baz;
  return 0;
}
```

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

CLASSES DEFINED WITH STRUCT AND UNION

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

The keyword ***struct***, generally used to declare plain data structures, can also be used to declare classes that have member functions, with the same syntax as with keyword class.
The only difference between both is that members of classes declared with the keyword **struct have public access by default**, while members of classes declared with the keyword **class have private access by default**. For all other purposes both keywords are equivalent in this context.

Conversely, the concept of ***unions*** is different from that of classes declared with struct and class, since **unions only store one data member at a time**, but nevertheless they are also classes and can thus also **hold member functions**. The default access in **union classes is public**.