# Classes (I)

OOP in C++

## 1. Introduction

**Class**: A class is a type of data structure that can hold data members and member functions.
**Object**: An instantiation of a class.
Classes are defined using the keyword `class` or `struct`. A typical class implementation may look like the following:

```
class ClassName {
accessSpecifier1:
  member1;
accessSpecifier2:
  member2;
    ...
} objectNames;
```

Access specifiers define access rights to the data stored within a class. They can be one of `private`, `public`, or `protected`. By default, members are set to `private`.

1. `private` members can only be accessed from within the class itself.
2. `public` members can be accessed anywhere.
3. `protected` members can be accessed from within the class and through derived classes.

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

In the code above, we create a Rectangle class and instantiate an object rect. We define a private width and length, and public methods `set_values` and `area`. We are able to access the public members simply as normal function calls:

```
rect.set_values(5, 4);
rect.area();
```

Note that the code above does not implement any behaviour to `set_values` nor `area`. We can either implement behaviour within the class as a normal function, or use the scope operator and define it outside, as follows:

```
#include <iostream>
```

```cpp

//create Rectangle Class
class Rectangle {
  int width, length;
public:
  void set_values(int, int);
  int area(void) {
    return width*length;
    }
};

//We can implement behaviour of prototype functions within a class by using the
↪  :: operator.
void Rectangle::set_values(int x, int y){
  width = x;
  length = y;
}

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

The code above will print the value `12`. Note that each instantiation of a class is independent to each other. For example,

```cpp
...
int main(){
  Rect rectA, rectB;
  rectA.set_values(3, 4);
  rectB.set_values(5, 6);
  std::cout << "rectA area: " << rectA.area() << std::endl;
  std::cout << "rectB area: " << rectB.area() << std::endl;
  return 0;
}
```

Output:

```
rectA area: 12
rectB area: 30
```

## 2. Constructors

In the code shown above, what would happen if we were to call `area()` before `set_values`? An undetermined result, since we haven't initialized width or height. In order to avoid this behaviour, we can introduce a special member, known as a constructor.
**Constructor**: A constructor is a method that is called automatically when a new object is instantiated.
A constructor is defined by a method that has the exact same name as the class, and no return type; not even `void`. For example:

```cpp
#include <iostream>

class Rectangle {
  int width, length;
public:
  Rectangle(int, int); //prototype constructor - takes the same name as the
  ↪  class, with no return type
  int area(void) {
    return width * length;
  }
};

//constructor definition
Rectangle::Rectangle(int x, int y){
  width = x;
  length = y;
}

int main(){
  //instantiate rect with (3, 4) parameters
  Rectangle rect(3, 4);
  std::cout << rect.area() << std::endl;
  return 0;
}
```

Output:

```
12
```

## 2.1. Overloading Constructors

Constructors also have the ability to be overloaded. This can be useful in cases where you want different instantiation behaviour. For example:

```cpp
#include <iostream>

class Rectangle {
  int width, height;
public:
  //Rectangle constructor with 2 parameters
  Rectangle(int x, int y){
    width = x;
    height = y;
  }

  //Rectangle constructor with 1 parameter
  Rectangle(int x){
    width = x;
    height = x;
  }

  //overload default constructor
  Rectangle(){
    width = 10;
    height = 10;
  }

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

int main() {
  Rectangle rect(3, 4);
  Rectangle square(5);
  Rectangle default_rect;
  return 0;
}
```

In this case, since `square` only has one parameter, it will assume the Rectangle constructor that only accepts one parameter. As well, we have overloaded the default constructor, which is called when we pass no parameters. Note that `default_rect()` would NOT work. In this case, it would create a `default_rect()` function declaration, instead of calling the default constructor.

### 2.1.1 Uniform initialization

The method of calling constructors as shown above, by enclosing the arguments in parenthesis is known as functional form. However, C++ allows for other syntaxes.

First, if a constructor only takes a single parameter, it can be called implicitly with variable assignment:

```
ClassName objectName = initializationValue;
```

The code above is equivalent to

```
ClassName objectName = initializationValue;
```

Added in C++11, uniform initialization is essentially the same as functional form, but instead uses braces.

```
ClassName objectName { value1, value2, ... }
```

The benefit of uniform initialization is that it strictly enforces types, which protects you from accidentally losing precision.

```
int x{3.14}; //ERROR: Narrowing conversion
int y(3.14); //Allowed, however y becomes 3.
```

For example:

```
#include <iostream>
const double PI = 3.14159265;

class Circle {
  double radius;
public:
  Circle(double r) {
    radius = r;
  }

  double circum() {
    return 2 * radius * PI;
  }
};

int main() {
  Circle foo(10.0); //Functional initialization
  Circle bar = 20.0; //Assignment initialization
  Circle baz {40.0}; //uniform initialization
  return 0;
}
```

Whilst this choice of constructors is largely just a matter of style, many newer style guides suggest to use uniform initialization.

## 2.2. Member Initialization in Constructors

Many times in a constructor, we need only to initialize other members. This can be done by using a colon and a list of initializations. For example,

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

Here, the constructor can be defined as shown before as

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

Or, using member initialization:

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

Notice that we have an empty constructor body as well. This is because we want the constructor to only initialize the members, and have no added functionality.

While default constructing may just seem like a convenient shorthand, it does have some applications where it is necessary. Consider the following:

```
#include <iostream>
const double PI = 3.14159265

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

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 << "Volume: " << foo.volume() << std::endl;
  return 0;
}
```

In this example, the class `Cylinder` has a member of type `Circle`. Because objects of class `Circle` can only be initialized with a parameter, `Cylinder` needs to invoke `Circle`'s constructor. This is only possible through a member initialization list.
Note that initialization lists can also be created using uniform initialization.

```
Cylinder::Cylinder(double r, double h) : base{r}, height{h} {} //with {} syntax
```

## 2.3. Pointers to Classes

Since classes are just an object type, we can define a pointer that points at a class type. For example,

```
Rectangle *pRect;
```

Creates a pointer to an object of class `Rectangle`.
Much like with built in datastructures, we can access members of an object directly from a pointer by using the arrow operator.

```
#include <iostream>

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

int main() {
  Rectangle obj(3, 4); //object instantiation
  Rectangle *foo, *bar, *baz;
  foo = &obj;
  bar = new Rectangle(5, 6);
  baz = new Rectangle[2]{{2, 5}, {3, 6}};

  cout << "obj area: " << obj.area() << std::endl;
  cout << "*foo area: " << foo->area() << std::endl;
  cout << "*bar area: " << (*bar).area() << std::endl;
  cout << "baz[0] area: " << baz[0].area() << std::endl;
  cout << "baz[1] area: " << *(baz + 1).area() << std::endl;

  delete bar;
  delete[] baz;
  return 0;
}
```

Output:

```
obj area: 12
*foo area: 12
*bar area: 30
baz[0] area: 10
baz[1] area: 18
```

## 3. Classes Defined with Struct and Union

Classes also be defined using the keyword `struct` and `union`.

The keyword `struct` can also be used to declare classes that have member functions, which have the exact same syntax as `class`, however members have `public` access by default.

The concept of unions, however, is quite different to that of `class` and `struct`. Unions can only have a single data member active at a time, but they can also hold multiple member functions. That is, members in a union will share a common memory address, with the size allocated being the largest member. The default access for union is `public`.