



## Lecture Two

# Introducing Classes

**Ref: Herbert Schildt, Teach Yourself C++, Third Ed<sup>n</sup> (Chapter 2)**



# Class

## Structure in C Program

```
typedef struct mystruct {  
    int stdID;  
    double CGPA;  
} student;  
  
student Karim;
```

## Structure in C++ Program

```
struct mystruct {  
    int stdID;  
    double CGPA;  
public:    //Is it required?  
    int getStdID();  
    void setStdID(int ID);  
    double getCGPA();  
    void setCGPA(double CGPA);  
};  
  
mystruct Karim;
```

● Do we really need methods to get and set data in structure?

● All members of a structure is public. So, we can write

```
Karim.stdID = 324;
```

## Class in C++ Program

```
class myclass {  
    int stdID;  
    double CGPA;  
public:    //Is it required?  
    int getStdID();  
    void setStdID(int ID);  
    double getCGPA();  
    void setCGPA(double CGPA);  
};  
  
myclass Karim;
```

● Do we really need methods to get and set data in class?

● All members of a class is private.



# Constructor Function

- In real problem, virtually every object requires some sort of *initialization*. **Constructor function** performs the tasks of *initialization*.
- A class's constructor is called each time an object of that class is created.
- A constructor function has the same name as the class of which it is a part and has no return type.

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

class myclass {
    int a;
public:
    myclass(); // Constructor
    void show();
};

myclass::myclass(){
    cout << "In constructor\n";
    a = 10;
}
```

```
void myclass::show(){
    cout << a;
}
```

```
int main(){
    myclass ob;

    ob.show();
    return 0;
}
```

**OUTPUT:**  
In constructor  
10



# Destruction Function

- This function is called when an object is destroyed.
- While working with object, some actions may be performed when an object is destroyed, e.g., freeing the memory allocated by the object.
- **Local objects** are destroyed when they go out of scope. **Global objects** are destroyed when the program ends.

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

class myclass {
    int a;
public:
    myclass(); // constructor
    ~myclass(); // destructor
    void show();
};

myclass::myclass(){
    cout << "In constructor\n";
    a = 10;
}
```

```
myclass::~~myclass(){
    cout << "Destructing.....\n";
}

void myclass::show(){
    cout << a << '\n';
}

int main(){
    myclass ob;

    ob.show();
    return 0;
}
```

## OUTPUT:

In constructor

10

Destructing.....



# Constructor that takes parameters

- Arguments can be passed to the constructor function.
- Unlike constructor functions, destructor functions cannot have parameters. There exists no mechanism by which to pass arguments to an object that is being destroyed.

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

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr); // constructor
    ~strtype(); // destructor
    void show();
};

strtype::~strtype(){
    cout << "Freeing p.....\n";
    free(p);
}

void strtype::show(){
    cout << p << " - length: " << len;
    cout << '\n';
}
```

```
strtype::strtype(char *ptr){
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if (!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    strcpy(p, ptr);
}

int main(){
    strtype s1("This is a test."), s2("I like C++");

    s1.show();
    s2.show();
    return 0;
}
```

## OUTPUT:

```
This is a test.-length: 15
I like C++-length: 10
Freeing p.....
Freeing p.....
```



# Introducing Inheritance

- Inheritance is the mechanism by which one class can inherit the properties of another.
- When one class is inherited by another, the class that is inherited is called the *base class*. The inheriting class is called the *derived class*.
- *Base class* represents the most *general description* of a set of traits. A *derived class* inherits those general traits and adds properties that are specific to that class.

```
// Define base class
```

```
class B {  
    int i;  
public:  
    void set_i(int n);  
    int get_i();  
};
```

```
// Define derived class
```

```
class D: public B {  
    int j;  
public:  
    void set_j(int n);  
    int mul();  
    int get_i();  
};
```

The keyword *public* tells that

- ✓ All the *public elements* of the base class will also be *public elements* of the derived class.
- ✓ All the *private elements* of the base class remain private to it and are *not directly accessible* by the derived class.



# Object Pointers

- When a pointer to the object is used, the arrow operator ( $\rightarrow$ ) is employed rather than dot (.) operator.
- Just like pointers to other types, an object pointer, when **incremented**, will point to the next object of its type.

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

class myclass {
    int a;
public:
    myclass(int x); // constructor
    int get();
};

myclass::myclass(int x){
    a = x;
}

int myclass::get(){
    return a;
}
```

```
int main(){
    myclass ob(120);
    myclass *p;

    p = &ob;
    cout << "Value using object: " << ob.get();
    cout << "\n";
    cout << "Value using pointer: " << p->get();

    return 0;
}
```



# Class, Structure and Union

## Class and Structure:

- The class and the structure have identical capabilities.
- In C++, the definition of structure has been expanded so that it can also include member functions including constructor and destructor.

## What is the difference between Class and Structure?

By default, the **members of a class are private** but the **members of a structure are public**.

## Why is the Duplication/Redundancy?

- ✓ Expanded definition of structure concerns maintaining upward compatibility from C.
- ✓ Class is syntactically separate entity from structure, so future direction of C++ is not **restricted by compatibility** concerns.





# Class, Structure and Union

## Union in C++:

- In C++, a union defines a class type that can contain both data and functions including constructor and destructor.
- Like structure, by default, all members of a union are public until the private specifier is used.
- Like union in C, all data members share the same memory location.
- C unions are upwardly compatible with C++ unions.

## How does union differ from class?

- ✓ In an object oriented language, it is important to preserve **encapsulation**. Union combines code and data together in which all data uses a shared location.
- ✓ Class does not allow different data to share same location.

## Several Restrictions that apply to Union in C++

- ✓ Unions **cannot inherit** any other class and they cannot be used as a base class for any other type.
- ✓ Unions must not have any static members
- ✓ Though union, itself, can have a constructor and destructor; they must not contain any object that has a constructor or destructor.
- ✓ Unions cannot have virtual member functions.



# Class, Structure and Union

## Anonymous Union:

- An anonymous union does not have a type name, and no variables can be declared for this sort of union.

```
union {           //an anonymous union
    int i;
    char ch[4];
};
```

- An anonymous union tells the compiler that its members will share the same memory location.
- Members of an anonymous union act and are treated like normal variable, i.e., the members are accessed directly, **without the dot operator**.
- Anonymous unions have all the restrictions that apply to the normal unions, plus these additions-
  - A **global anonymous** union must be declared **static**.
  - An anonymous union **cannot contain private** members.
  - The names of the members of an anonymous union must **not conflict with other identifiers** within the same scope.



# In-line function

- 🌐 In-line functions are not actually called, rather are expanded in line, at the point of each call. (like macros in C).
- 🌐 **Advantage:** in-line function has no overhead associated with the function call and return mechanism, so much faster than the normal function calls.
- 🌐 **Disadvantage:** If in-line functions are too large and called too often, program grows larger. Therefore, only short functions are declared in-line.

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

inline int even(int x){
    return !(x%2);
}
```

```
int main(){
    if (even(10)) cout << "10 is even.\n";
    return 0;
}
```

## Advantages of in-line functions over parameterized macros:

- ✓ More **structure way to expand** short function in line, e.g., in parameterized macros it is easy to forget that extra parenthesis are often needed to ensure proper in-line expansion and using in-line function prevent this problems.
- ✓ An in-line function might be able to be optimized more thoroughly by the compiler.

**Inline specifier is a request, not a command to the compiler:** Some compiler will not inline a function if it contains a static variable, loop, switch or goto.

If any inline restriction is violated, the compiler is free to generate a normal function.



# Automatic In-line function

- If a member's function definition is short enough, the definition can be included inside the class declaration. Doing so causes the function to automatically become an in-line function.
- When a function is defined within a class declaration, the **inline** keyword is no longer necessary.

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

class samp {
    int i, j;
public:
    samp( int a, int b);
    int divisible() { return !(i%j);}
};

samp::samp(int a, int b){
    i = a;
    j = b;
}
```

```
int main(){
    samp ob1(10, 2), ob2(10, 3);

    if (ob1.divisible()) cout << "10 is divisible by 2\n";
    if (ob2.divisible()) cout << "10 is divisible by 3\n";

    return 0;
}
```

- When a function defined inside a class declaration cannot be made into in-line function, it is automatically made into a regular function..
- From the compiler's point of view, there is no difference between the compact style and the standard style, however, compact style is commonly used in C++ programs.
- The same restriction that apply to a normal in-line functions apply to automatic in-line functions within a class declaration.