

Object Oriented Programming Using C++

Ketan G Kore

Ketan.kore@sunbeaminfo.com

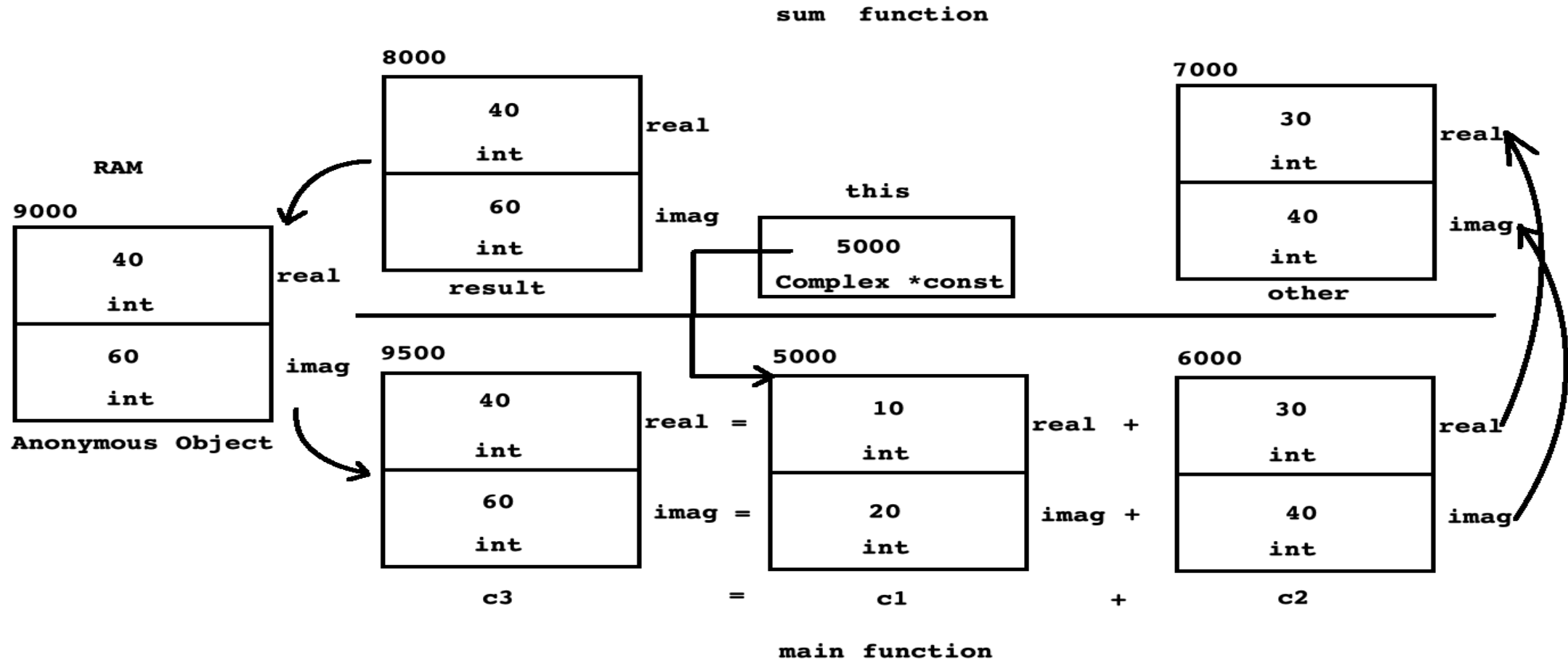
this pointer

- **Consider steps in program development**
 1. Understand clients requirement i.e. **problem statement**.
 2. Depending on the **requirement**, define class and declare data member inside it.
 3. Create object of the class.(Here data member will get space inside object)
 4. To process state of the object, call member function on object and define member function inside class.
- If we call **member function** on object then compiler **implicitly pass, address of current object** as a **argument** to the **function**.
- To store address of **argument(current object)**, compiler **implicitly declare one pointer** as **function parameter inside member function**. It is called **this pointer**.
- General type of this pointer is : **ClassName *const this;**
- **this** is a keyword in C++.

this pointer

- Using this pointer, non static data member and non static member function can communicate with each other hence, it is considered as a link/connection between them.
- If name of the data member and name of local variable is same then preference is always given to the local variable. In this case to refer data member, we must use this keyword. Otherwise use of this keyword is optional.
- this pointer is implicit pointer, which is available in every non static member function of the class which is used to store address of current object or calling object.
- Following member function do not get this pointer:
 1. Global function
 2. Static member function
 3. Friend function
- We can not declare this pointer explicitly. But compiler consider this pointer as a first parameter inside member function.

this pointer



- We can create object without name. It is called anonymous object.
- If we return object from function by value then compiler implicitly generates anonymous object in RAM.

Function Overloading

- If implementation of function is logically same/equivalent then we should give same name to the function.
- Using following rules we can give same name to the function.

1. If we want to give same name to the function and if type of all the parameters are same then number of parameters passed to the function must be different.

```
void sum( int num1, int num2 ){ //2 parameters
    int result = num1 + num2;
    cout<<"Result : " << result << endl;
}

void sum( int num1, int num2, int num3 ){ //3 parameters
    int result = num1 + num2 + num3;
    cout<<"Result : " << result << endl;
}
```

Function Overloading

2. If we want to give same name to the function and if number of parameters are same then type of at least one parameter must be different.

```
void sum( int num1, int num2 ){  
    int result = num1 + num2;  
    cout<<"Result : "<<result<<endl;  
}  
  
void sum( int num1, double num2 ){  
    double result = num1 + num2;  
    cout<<"Result : "<<result<<endl;  
}
```

Function Overloading

3. If we want to give same name to the function and if number of parameters are same then order of type of parameters must be different.

```
void sum( int num1, float num2 ){  
    float result = num1 + num2;  
    cout<<"Result : "<<result<<endl;  
}  
  
void sum( float num1, int num2 ){  
    float result = num1 + num2;  
    cout<<"Result : "<<result<<endl;  
}
```

Function Overloading

4. Only on the basis of **different return type**, we can **not give same name to function**.

- Using above rules, **Process of defining multiple functions with name** is called **function overloading**. In other words, process of defining function with **same name and different signature** is called function overloading.
- Functions, which are **taking part in function overloading** are called **overloaded function**
- Function overloading is **OOPS concept**. It represents compile time **polymorphism**.

Function Overloading

- If implementation of function is logically same/equivalent then we should overload function.
- For function overloading, functions must exist inside same scope.
- **In function overloading return type is not considered.**
 - Returning value from function and catching value which returned by function is optional hence we can not overload function.
- Except 2 functions, we can overload any global function as well as member function:
 1. main function
 2. destructor

Function Overloading

Mangled name

- If we define function in C++ then by looking toward name of the function and type of parameters passed to the function compiler implicitly generates unique name. It is called mangled name.

```
void sum( int num1, int num2 ){ //__Z3sumii
    //TODO
}
void sum( int num1, float num2 ){ //__Z3sumif
    //TODO
}
void sum( int num1, float num2, double num3 ){ //__Z3sumifd
    //TODO
}
```

- Mangled name may vary from compiler to compiler.

Name Mangling

- To generate mangled name, compiler implicitly use one algorithm. It is called name mangling.

Function Overloading

```
void print( int number ){
    cout<<"int   :   "<<number<<endl;
}
void print( float number ){
    cout<<"float   :   "<<number<<endl;
}
int main( void ){
    //::print(10); //int   :   10
    //::print( 10.5f ); //float :   10.5
    ::print( 10.5 ); //error: call to 'print' is ambiguous
    return 0;
}
```

Default Arguments

- In C++, we can assign default value to the parameter of function. Such default value is called default argument and parameter is called optional parameter.

```
void sum( int num1, int num2, int num3 = 0, int num4 = 0, int num5 = 0 ){  
    int result = num1 + num2 + num3 + num4 + num5;  
    cout<<"Result    :    "<<result<<endl;  
}  
  
int main( void ){  
    sum( 10, 20 );  
    sum( 10, 20, 30 );  
    sum( 10, 20, 30, 40 );  
    sum( 10, 20, 30, 40, 50 );  
    return 0;  
}
```

Default Arguments

- Default arguments are always assigned from right to left direction.
- Using default argument, we can reduce developers effort.
- We can assign default arguments to the parameters of any global function as well as member function.
- In case of modular approach, default arguments must appear in declaration part(.h)

```
void sum( int num1, int num2, int num3 = 0, int num4 = 0, int num5 = 0 );  
int main( void ){  
    //TODO : Call sum function  
    return 0;  
}  
void sum( int num1, int num2, int num3, int num4, int num5){  
    //TODO : add number  
}
```

Initialization

- During declaration of variable, process of providing value to the variable is called initialization.
- We can initialize any variable only once.

```
int num1 = 10;    //OK : Initialization  
int num1 = 20;    //Not Ok  
int num2 = num1;   //OK : Initialization
```

Assignment

- After declaration of variable, process of providing value to the variable is called assignment.
- We can assign value to the variable multiple times.

```
int num1 = 10;  //OK : Initialization  
num1 = 20;    //OK : Assignment  
num1 = 30;    //OK : Assignment
```

```
int num1;  //OK  
num1 = 10; //OK : Assignment  
num1 = 20; //OK : Assignment
```

Constructor

- It is a member function of the class which is used to initialize the object.
- Due to following reasons, constructor is considered as special member function of a class:
 1. Its name is same as class name.
 2. It doesn't have any return type.
 3. It gets called implicitly
 4. It get called once per object.
- We can not call/invoke constructor on object/pointer/reference explicitly. It is designed to call implicitly.

Constructor

```
Complex c1; //c1 => Object  
//c1.Complex( ); //Not OK  
  
Complex *ptr = &c1; //ptr => Pointer  
//ptr->Complex( ); //Not OK  
  
Complex &c2 = c1; //c2 => reference  
//c2.Complex( ); //Not OK
```

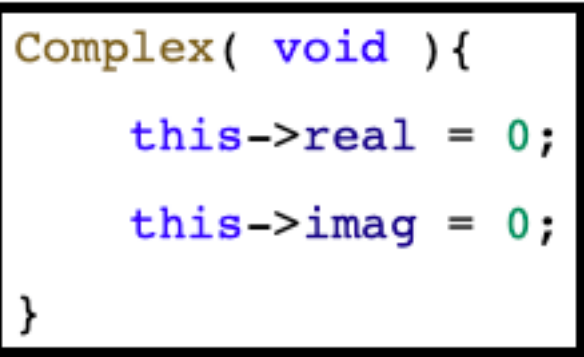
Constructor

- We can not declare constructor static, constant, volatile or virtual. **We can declare constructor inline only.**
- We can use any access specifier on constructor.
 1. If constructor is public then we can create object of the class inside member function as well as non member function.
 2. If constructor is private then we can create object of the class inside member function only.
- **Constructor** calling sequence depends on order of object declaration.
- Types of constructor
 1. Parameterless constructor
 2. Parameterized constructor
 3. Default constructor.

Parameterless Constructor

- We can define constructor without parameter(parameterless). It is called parameterless constructor.

```
Complex( void ){  
    this->real = 0;  
    this->imag = 0;  
}
```

- passing argument then parameterless constructor gets called.
 - Complex c1; //Here on c1, parameterless constructor will call.
- It is also called as zero argument constructor or user defined default constructor.

Parameterized Constructor

- We can define constructor with parameter. It is called parameterized constructor.

```
Complex( int real, int imag ){  
    this->real = real;  
    this->imag = imag;  
}
```

- If we create object, by passing argument then parameterized constructor gets called.
 - Complex c1(10,20); //Here on c1, parameterized constructor will call.

Default Constructor

- If we do not define constructor inside class then compiler generate one constructor for that class by default. It is called default constructor.
- Compiler generated default constructor is parameterless constructor. In other words, if we want to create object by passing arguments then we must define parameterized constructor inside class.

```
class Complex{  
public:  
    /*  
        Complex( void ){  
            //Empty  
        }  
    */  
};
```

Constructor's Member_INITIALIZER List

- If we want to initialize **data members**, according to order of data member declaration then we should use **Constructor's member initializer list**.

```
class Test{
    int num1, num2, num3;
public:
    Test( void ) : num1( 10 ), num2( 20 ), num3( 30 ){
    }
    Test( int num1, int num2, int num3 ) : num1( num1 ), num2( num2 ), num3( num3 ){
    }
};
```

- Except **array and string** we can initialize any data member in Constructor member initializer list.
- In case of modular approach, Constructor member initializer list must appear in definition part.

Thank you