Department of Computer Science and Engineering

PES UNIVERSITY

Object Oriented Programming with C++
UE22CS221A

UNIT 2

Ms. Kusuma K V

Structure and Class, Data Member, Member Function, Access Specifier

Structures combine logically related data items into a single unit. The data items enclosed within a structure are known as members and they can be of the same type or different types. Hence, a structure can be viewed as a heterogeneous user defined data type. It can be used to create variables which can be manipulated in the same way as variables of standard types. It encourages better organization and management of data in a program. (In C, structures cannot have functions within them). **In C++, structures can have both data and functions as its members.**

```
typedef struct structName
{
        Access specifier:    //can be private, public or protected. public by default
        Data members                //attributes
        Member functions            //methods to access data members
        //Note: structure definition ends with a semi colon
}typeName;
```

where, struct is keyword

typedef is a keyword used to give alias name to the structure

structName is the used defined structure name.

typeName is the alias name to **struct StructName.**

In C++ we define our own data structures by defining a **class**. A class defines a type along with a collection of operations that are related to that type. The class mechanism is one of the most important features in C++. In fact, a primary focus of the design of C++ is to make it possible to define class types that behave as naturally as the built-in types.

```
class ClassName
{
        Access specifier:   //can be private, public or protected. private by default
        Data members                //attributes
        Member functions            //methods to access data members
        //Note: class definition ends with a semi colon
};
```

where, class is keyword and ClassName is the user defined name given for class.

Note: In C++, a class defined with the class keyword has private members and base classes by default. A structure is a class defined with the struct keyword. Its members and base classes are public by default. In practice, structs are typically reserved for data without functions.

- A class is an implementation of a type. It is the way to implement User-defined Data Type (UDT)
- A class contains data members / attributes
- A class has operations / member functions / methods
- Classes offer data abstraction, encapsulation of Object Oriented Programming
- Classes are similar to structures that aggregate data logically
- A class is defined by class keyword
- Classes provide access specifiers for members to enforce data hiding that separates implementation from interface
    - private - accessible inside the definition of the class
    - public - accessible everywhere
    - protected - accessible inside the definition of the class and derived classes
- A class is a blue print for its instances (objects)

//structComplex_In_C.c

```
/*Complex number using structures in C*/
/*C structure cannot have member functions*/
#include<stdio.h>

typedef struct Complex
{
        int re;                         //real part
        int im;                         //imaginary part
}Complex;

void showComplex(Complex c)
{
        printf("Complex number=%d+%di\n",c.re,c.im);
}

int main()
{
        Complex c1;                     //c1 is variable of type struct Complex
        c1.re=2;
        c1.im=3;
        showComplex(c1);
```

```
    Complex c2;                //c2 is variable of type struct Complex
    c2.re=4;
    c2.im=5;
    showComplex(c2);
    return 0;
}
```

```
/*Output:
Complex number=2+3i
Complex number=4+5i
*/
```

Object

An object of a class is an instance created according to its blue print. Objects can be automatically, statically, or dynamically created

- An object comprises data members that specify its state
- An object supports member functions that specify its behavior
- Data members of an object can be accessed by "." (dot) operator on the object
- Member functions are invoked by "." (dot) operator on the object
- An implicit this pointer holds the address of an object.
- This serves the identity of the object in C++
- this pointer is implicitly passed to methods

Access Specifier

- Classes provide access specifiers for members (data as well as function) to enforce data hiding that separates implementation from interface
- private - accessible inside the definition of the class
    - member functions of the same class
- public - accessible everywhere
    - member functions of the same class
    - member function of a different class
    - global functions
- The keywords public, private, and protected are the Access Specifiers
- Unless specified, the access of the members of a class is considered private
- A class may have multiple access specifier. The effect of one continues till the next is encountered

Information Hiding

- The private part of a class (attributes and methods) forms its implementation because the class alone should be concerned with it and have the right to change it.
- The public part of a class (attributes and methods) constitutes its interface which is available to all others for using the class.
- Customarily, we put all attributes in private part and the methods in public part. This ensures:
    - The state of an object can be changed only through one of its methods (with the knowledge of the class).
    - The behavior of an object is accessible to others through the methods
- This is known as Information Hiding

this pointer

- An implicit this pointer holds the address of an object.
- this pointer serves as the identity of the object in C++.
- Type of this pointer for a class X object:
  X * const this;
- this pointer is accessible only in methods.
- this pointer is implicitly passed to methods.

Before compilation	After compilation

class X {
void f(int, int);
...
};
	void X::f(**X * const this**, int, int)
	{
	
	};

X a;
a.f(2, 3);
	X::f(**&a**, 2, 3); // &a = this

```
//structComplex1_In_CPP.cpp
/*Complex number using structures in C++
*to demonstrate backward compatibility with C
*only changes made to structComplex.c is wrt the I/O
*Program compiles and runs successfully because
*all data members and member functions are public by default
*/
#include<iostream>
```

```
using namespace std;

typedef struct Complex
{
	int re;
	int im;
}Complex;

void showComplex(Complex c)
{
	cout<<"Complex number="<<c.re<<"+"<<c.im<<"i"<<endl;
}

int main()
{
	Complex c1;
	c1.re=2;
	c1.im=3;
	showComplex(c1);

	Complex c2;
	c2.re=4;
	c2.im=5;
	showComplex(c2);
	return 0;
}
/*Output:
Complex number=2+3i
Complex number=4+5i
*/
```

```
//structComplex2_In_CPP.cpp
/*Complex number using structures in C++
* in C++, structures can have member functions along with member data
* all data members and member functions are public by default
* Notice the use of scope resolution operator(::),used to define member functions outside the class
*/
#include<iostream>
```

```
using namespace std;

typedef struct Complex
{
        int re;
        int im;

        void readComplex(int r,int i)
        {
        re=r;
        im=i;
        }
        void showComplex();
}Complex;

//Complex class has 2 member functions: readComplex and showComplex
//readComplex is defined inside the structure
//showComplex is defined outside the structure using class name and scope
//resolution operator
//Notice that showComplex has to be declared inside the structure
void Complex::showComplex()
{
        cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}
int main()
{
        Complex c1;
        c1.re=2;            //data member re is assigned the value 2
        c1.im=3;            //data member im is assigned the value 3
        c1.showComplex();   //showComplex is called for c1object
//notice that data members are accessed using object and .(dot) operator
//notice that member functions are called using object and .(dot) operator
        Complex c2;
        c2.readComplex(4,5);        //readComplex member function is used to assign
                                    //values to members of c2 object
        c2.showComplex();
        return 0;
}
/*Output:
```

```
Complex number=2+3i
Complex number=4+5i
*/
//classComplex1.cpp          //This pgm doesn't compile
/*Complex number using class in C++
* all data members and member functions are private by default
* private members are accessible only within the class
*/

/*Program doesn't compile, because all members are private and therefore cannot
be accessed outside the class
* main function(which is outside the class) is trying to access the private data
members and member function
* Look at classComplex2.cpp: the default private access is changed to public access
*/
#include<iostream>
using namespace std;

class Complex
{
        int re;
        int im;

        void readComplex(int r,int i);
        void showComplex();
};

void Complex::readComplex(int r,int i)
{
        re=r;
        im=i;
}

void Complex::showComplex()
{
        cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}
```

```
/*main() is trying to access private members(data and function), therefore the code doesn't compile*/
int main()
{
	Complex c1;
	c1.re=2;
	c1.im=3;
	c1.showComplex();

	Complex c2;
	c2.readComplex(4,5);
	c2.showComplex();
	return 0;
}
```

//classComplex2.cpp

```
/*Complex number using class in C++
*all data members and member functions are made public
*public members are accessible both inside and outside the class
*program compiles and runs successfully, but there is no security for the data
*i.e., anyone can make changes to the data, there is no controlled access
*Look at complexClass3.cpp which provides controlled acess to data members
*/

/*this ptr: Constant pointer associated with non-static member functions of the class
*An implicit this pointer holds the address of an object. This serves as the identity of the object in C++
*It always points at the object w.r.t which function it is called
*this pointer is implicitly passed to methods
*Compiler puts declaration of this pointer as a leading formal argument in the prototype of all member functions
*Compiler puts definition of this pointer as a leading formal argument in the definition of all member functions
*Compiler passes the address of the invoking object as a leading parameter to each call to the member functions
*Here shown as comment for showComplex member function
*/
#include<iostream>
using namespace std;
```

```
class Complex
{
public:
        int re;
        int im;

        void readComplex(int r,int i)
        {
                re=r;
                im=i;
        }

        void showComplex();        // After Compilation: void showComplex(Complex* const);
};

void Complex::showComplex()
{
        cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}

/*After Compilation:
void Complex::showComplex(Complex* const this)
{
        cout<<"Complex number="<<this->re<<"+"<<this->im<<"i"<<endl;
}
*/

int main()
{
        Complex c1;
        c1.re=2;
        c1.im=3;
        c1.showComplex();          //After Compilation: showComplex(&c1);
        Complex c2;
        c2.readComplex(4,5);
        c2.showComplex();          //After Compilation: showComplex(&c2);
        return 0;
}
/*
```

```
Output:
Complex number=2+3i
Complex number=4+5i
*/

//classComplex3.cpp
/*Complex number using class in C++
*all data members are made private and member functions are made public
*i.e., only the public member functions of the class can acess the data
*there by providing controlled access to the data
*member functions defined inside the class are inline
*member functions defined outside the class can be made inline
*by prefixing the keyword inline either in the function prototype(see readComplex)
*or in the function definition(see showComplex)
*/

#include<iostream>
using namespace std;

class Complex
{
private:
        int re;
        int im;
public:
        inline void readComplex(int r,int i);
        void showComplex();
};
void Complex::readComplex(int r,int i)
{
        re=r;
        im=i;
}
inline void Complex::showComplex()
{
        cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}
int main()
{
	Complex c1;
//	c1.re=2;	/*re is private therefore can be accessed only by public member fns and friend (discussed later)*/
//	c1.im=3;	/*im is private therefore can be accessed only by public member fns and friend (discussed later)*/
	c1.readComplex(2,3);
	c1.showComplex();
	Complex c2;
	c2.readComplex(4,5);
	c2.showComplex();
	return 0;
}
/*Output:
Complex number=2+3i
Complex number=4+5i
*/
```

//classComplex4.cpp

```
/*Complex number using class in C++
* all data members are made private and member functions are made public
* i.e., only the public member functions of the class can acess the data
* there by providing controlled access to the data
* notice the use of getter(accessor) and setter(mutator) methods
*/
#include<iostream>
using namespace std;
class Complex
{
private:
	int re;
	int im;
public:
	int getRE()
	{
		return re;
	}
	int getIM()
	{
		return im;
	}
	void setRE(int r)
	{
		re=r;
	}
	void setIM(int i)
	{
		im=i;
	}
};
int main()
{
	Complex c1;
	c1.setRE(2);
	c1.setIM(3);
	cout<<"Complex number1:"<<c1.getRE()<<"+"<<c1.getIM()<<"i"<<endl<<endl;

	Complex c2;
	int r,i;
	cout<<"Enter the second complex number"<<endl;
	cout<<"Real part:";
	cin>>r;
	c2.setRE(r);
	cout<<"Imaginary part:";
	cin>>i;
	c2.setIM(i);
	cout<<endl<<"Complex number2:"<<c2.getRE()<<"+"<<c2.getIM()<<"i"<<endl;
	return 0;
}
/* Output:
Complex number1:2+3i

Enter the second complex number
Real part:4
Imaginary part:5

Complex number2:4+5i
*/
```

//classComplex5.cpp

```
/*Complex number using class in C++
*constructor: special member function which has the name same as that of the class
but doesn't have any return type
*constructors are called immediately after the birth of an (instance of a class)  i.e.,
*after memory is allocated for the object
*if we do not provide a constructor then compiler inserts a default constructor in the
code which is called during the process of object creation
*default constructor doesn't initialize members to default values
*provide a constructor manually as shown in classComplex6.cpp
*/
#include<iostream>
using namespace std;

class Complex
{
private:
        int re;
        int im;
public:
        void readComplex(int r,int i)
        {
        re=r;
        im=i;
        }
        void showComplex();
};

void Complex::showComplex()
{
        cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}
int main()
{
        Complex c1;   //default constructor doesn't initialize members to default zero
        c1.showComplex();             //displays garbage value

        c1.readComplex(2,3);          //assigning values to c1 object
        c1.showComplex();             //displays values of c1 object
        Complex c2;
        c2.readComplex(4,5);
        c2.showComplex();
        return 0;
}
/*Output:
Complex number=4196992+0i
Complex number=2+3i
Complex number=4+5i
*/
```

Constructor (ctor)

Each class defines how objects of its type can be initialized. Classes control object initialization by defining one or more special member functions known as constructors. The job of a constructor is to initialize the data members of a class object. A constructor is run whenever an object of a class type is created.

Constructors have the same name as the class. Unlike other functions, constructors have no return type. Like other functions, constructors have a (possibly empty) parameter list and a (possibly empty) function body. A class can have multiple constructors. Like any other overloaded function, the constructors must differ from each other in the number or types of their parameters.

The Synthesized Default Constructor

Classes control default initialization by defining a special constructor, known as the default constructor. The default constructor is one that takes no arguments.

If our class does not explicitly define any constructors, the compiler will implicitly define the default constructor for us.

The compiler-generated constructor is known as the synthesized default constructor. For most classes, this synthesized constructor initializes each data member of the class as follows:

- If there is an in-class initializer , use it to initialize the member(C++ 11 onwards).
- Otherwise, default-initialize the member(garbage value in this case).

//In the below program user has not defined ctor explicitly, compiler defined synthesized ctor initializes to the in-class initializer values.

//cPP11DefaultInit.cpp

```
#include<iostream>
using namespace std;
class Demo
{
        int a=2;                //in-class initialization, c++ 11 feature
        const int b=3;            //in-class initialization, c++ 11 feature

public:
        void show(){cout<<"a="<<a<<" b="<<b<<endl;}
};
int main()
{
        Demo d;  //Notice that default constructor initializes with the default value, not garbage
        d.show();
        return 0;
}
//compilation on g++:   g++ -std=c++11 cPP11DefaultInit.cpp
/*Output:
a=2 b=3
*/
```

```
//cPPDefaultCtor.cpp
#include<iostream>
using namespace std;
class Demo
{
        int a;
public:
        void show(){cout<<"a="<<a<<endl;}
};
int main()
{
        Demo d;  // default constructor is called and initializes the data member with garbage value
        d.show();
        return 0;
}
//Note, garbage initialization of data member a.
/*Output
a=-1505329600
*/
```

Note: The compiler generates a default constructor automatically only if a class declares no constructors.

Constructor Initializer List, Parameterized ctor

Look in to the ctor definition below(class Demo): the colon and the code between it and the curly braces define the (empty) function bodies. This new part is a constructor initializer list, which specifies initial values for one or more data members of the object being created. The constructor initializer is a list of member names, each of which is followed by that member's initial value in **parentheses** (or **inside curly braces, c++ 11 feature**). Multiple member initializations are separated by commas.

```
//memberInitDemo.cpp
#include<iostream>
```

```
using namespace std;
class Demo
{
	int m_a;
	int m_b;
public:
	void show(){cout<<"a="<<m_a<<" b="<<m_b<<endl;}
	Demo(int a,int b):m_a(a),m_b{b}{/*empty ctor body*/}
//m_a(a) inititializes m_a data member with the parameter a value
//m_b{b} initializes m_b data member with the parameter b value, c++ 11 feature (curly braces used for initialization)
};
int main()
{
	Demo d(2,3);  // parameterized constructor is called
	d.show();
	return 0;
}
/*Output: a=2 b=3*/
```

In the above eg, we see parameterized ctor **Demo(int a,int b)** which takes 2 parameters a and b.
Note: m_b{b} is valid only from c++ 11 onwards.
Note: do not use = in initialization list. Its an error.
For eg: Demo(int a,int b)**:m_a=a,m_b{b}**{/*empty ctor body*/}

Note: **Members are initialized in the order they are declared in the class.**

//memberInitOrder.cpp

```
//Order of initialization does not depend on the order in the initialization list.
//It depends on the order of data members in the class definition
#include<iostream>
using namespace std;

class OrderDemo
{
	int a;		//first declared member
	int b;		//second declared member

public:
	OrderDemo(int b1):b(b1),a(b){}
	void show(){cout<<"a="<<a<<" b="<<b<<endl;}
};
int main()
```

```
{
	OrderDemo o1(2);
	o1.show();
	return 0;
}
/*Output:
a=32764 b=2
*/
```

Note: a is initialized to garbage value, because in the class OrderDemo, first declared member is a, second declared member is b. So, **OrderDemo(int b1):b(b1),a(b){}**here first **a(b)**is executed which initializes a with b value(which is garbage as of now). Then, **b(b1)** is executed which initializes b with b1 balue(which is 2).

//memberInitOrder2.cpp

```
#include<iostream>
#include<cstring>
using namespace std;
class String
{
	int len;
	char *str;
public:
	String(char* s):str(s),len(strlen(str)){}
	void show(){cout<<"String is "<<str<<" Length="<<len<<endl;}
	~String(){cout<<"dtor called";}			//Dtor, explained in a while
};
int main()
{
	String s1("C++");
	s1.show();
	return 0;
}
/*Output: Segmentation fault (core dumped)*/
```

The below program gives proper output:

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

class String
{
	char *str;
	int len;
public:
	String(char* s):str(s),len(strlen(str)){}
	void show(){cout<<"String is "<<str<<" Length="<<len<<endl;}
	~String(){cout<<"dtor called"<<endl;}			//Dtor, explained in a while
};
int main()
{
	String s1("C++");
	s1.show();
	return 0;
}
/*Output:
String is C++ Length=3
dtor called
*/
```

Need of Member Initialization List:
To initialize constant data members, to initialize reference variables, etc.

//memberInitNeed.cpp

```
//Constant variables to be initialized during creation itself
#include<iostream>
using namespace std;
class Demo
{
	const int b;

public:
	Demo():b(2){}			//Initializes b with 2
//Below constructor is an error
//	Demo(){b=3;}
	void show(){cout<<"b="<<b<<endl;}
};
int main()
{
	Demo d;
	d.show();
	return 0;
}
/*Output:
b=2
*/
```

Defining a Constructor outside the Class Body (should be declared inside class)

Constructors have no return type, so this definition starts with the name of the function we are defining(which is class name itself). As with any other member function, when we define a constructor outside of the class body, we must specify the class of which the constructor is a member.

```
#include<iostream>
using namespace std;
class Demo
{
	int m_a;
	int m_b;
public:
	void show(){cout<<"a="<<m_a<<" b="<<m_b<<endl;}
	Demo(int a,int b);	//Declaration
};
//Defining ctor outside body using class name and scope resolution operator
Demo::Demo(int a,int b):m_a(a),m_b(b){/*empty constructor body*/}

/*Same as above definition
Demo::Demo(int a,int b):m_a(a),m_b(b)
{
	/*empty constructor body*/
}
*/

int main()
{
	Demo d(2,3);  // parameterized constructor is called
	d.show();
	return 0;
}
/*Output: a=2 b=3*/
```

If member initialization list is not used, then members may be assigned values in ctor body.

```
#include<iostream>
using namespace std;
class Demo
{
	int m_a;
	int m_b;
public:
	void show(){cout<<"a="<<m_a<<" b="<<m_b<<endl;}
	Demo(int a,int b);	//Declaration
};
```

```
//Defining ctor outside body using class name and scope resolution operator
Demo::Demo(int a,int b)
{
        m_a=a;
        m_b=b;
}

/*Same as above code, written in a single line instead of multiple lines
Demo::Demo(int a,int b) { m_a=a; m_b=b; }   */
int main()
{
        Demo d(2,3);  // parameterized constructor is called
        d.show();

        return 0;
}
```

//classComplex6Constructor.cpp

```
/*Complex number using class in C++
*default constructor doesn't initialize members to default values
*if we provide our own constructor then we cannot use the constructor provided by compiler
*automatic object creation
*see classComplex7Constructor.cpp to see how ctor's can be overloaded
*/

#include<iostream>
using namespace std;


class Complex
{
private:
        int re;
        int im;
public:
        Complex(int r,int i)
        {
                re=r;
                im=i;
        }

        void showComplex();
};

void Complex::showComplex()
{
	cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}


int main()
{
//	Complex c1;	//Error: because user has defined ctor, so default ctor is not invoked
	Complex c1(2,3);	//Calls user defined ctor in obj creation process
	c1.showComplex();

	Complex c2(4,5);
	c2.showComplex();

	return 0;
}

/*
Output:
Complex number=2+3i
Complex number=4+5i
*/
```

//classComplex7Constructor.cpp

```
/*Complex number using class in C++
*default constructor doesn't initialize members to default values
*if we provide our own constructor then we cannot use the constructor provided by compiler
*overload ctors: parameterized and zero argument(default)
*parameterizeed ctor is written using member initializer list
*(in classComplex6Constructor.cpp we had used assignment inside ctor body)
*automatic object creation
*/
#include<iostream>
using namespace std;

class Complex
{
private:
	int re;
	int im;
public:
//parameterized ctor
/*	Complex(int r,int i)
	{
		re=r;
		im=i;
	}
*/
//parameterized ctor using member initialization list
	Complex(int r,int i):re(r),im(i){}

//	Complex(int r,int i):re{r},im{i}{}	//C++ 11 feature initialization {}

//zero argument ctor or default ctor
	Complex(){re=0;im=0;}


	void showComplex();
};


void Complex::showComplex()
{
	cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}


int main()
{
	Complex c1;		//Calls zero argument ctor in obj creation process
	c1.showComplex();
	Complex c2(2,3);	//Calls parameterized ctor in obj creation process
	c2.showComplex();
	Complex c3(4,5);	//Calls parameterized ctor in obj creation process
	c3.showComplex();
	return 0;
}

/*Output:
Complex number=0+0i
Complex number=2+3i
Complex number=4+5i
*/

//classComplex8Constructor.cpp
/*Complex number using class in C++
*default constructor doesn't initialize members to default values
```

```
*if we provide our own constructor then we cannot use the constructor provided by
compiler
*overload ctors: parameterized ctor with default arguments, object can be created with
zero,one or two arguments
*Single constructor handles all 3 cases, because of default arguments
*automatic object creation
*/
#include<iostream>
using namespace std;

class Complex
{
private:
        int re;
        int im;
public:
//parameterized ctor
/*      Complex(int r=0,int i=0)
        {
                re=r;
                im=i;
        }
*/
//parameterized ctor using member initialization list and default arguments
        Complex(int r=0,int i=0):re(r),im(i){}
        void showComplex();
};
void Complex::showComplex()
{
        cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}
int main()
{
        Complex c1;
        c1.showComplex();     //Calls zero argument ctor in obj creation process
        Complex c2(2,3);      //Calls parameterized ctor in obj creation process
        c2.showComplex();

        Complex c3(4,5);      //Calls parameterized ctor in obj creation process
        c3.showComplex();
        return 0;
}

/*Output:
Complex number=0+0i
Complex number=2+3i
Complex number=4+5i
*/
```

Destructor (dtor)

Destructor is a special member function that cleans up an object when the object goes out of scope or is deleted. Just as a constructor controls initialization, the destructor controls what happens when objects of that class type are destroyed. Destructors do whatever work is needed to free the resources used by an object and destroy the nonstatic data members of the object.

The destructor is a member function with the name of the class prefixed by a tilde (~). It has no return value and takes no parameters:

```
class Demo{
public:
~Demo();    // destructor
// ...
};
```

Because it takes no parameters, it cannot be overloaded. There is always only one destructor for a given class.

What a Destructor Does

Just as a constructor has an initialization part and a function body, a destructor has a function body and a destruction part. In a constructor, members are initialized before the function body is executed, and members are initialized in the same order as they appear in the class. In a destructor, the function body is executed first and then the members are destroyed. **Members are destroyed in reverse order from the order in which they were initialized.**

The function body of a destructor does whatever operations the class designer wishes to have executed subsequent to the last use of an object. Typically, the destructor frees resources an object allocated during its lifetime.

In a destructor, there is nothing akin to the constructor initializer list to control how members are destroyed; the destruction part is implicit. What happens when a member is destroyed depends on the type of the member. Members of class type are destroyed by running the member's own destructor. The built-in types do not have destructors, so nothing is done to destroy members of built-in type.

When a Destructor Is Called

The destructor is used automatically whenever an object of its type is destroyed:

- Variables are destroyed when they go out of scope.
- Members of an object are destroyed when the object of which they are a part is destroyed.
- Elements in a container—whether a library container or an array—are destroyed

when the container is destroyed.

• Dynamically allocated objects are destroyed when the delete operator is applied to a pointer to the object.

• Temporary objects are destroyed at the end of the full expression in which the temporary was created.

Because destructors are run automatically, our programs can allocate resources and (usually) not worry about when those resources are released.

Note: The destructor is not run when a reference or a pointer to an object goes out of scope.

```
#include<iostream>
using namespace std;
class Demo
{
        int m_a;
        int m_b;
public:
        void show(){cout<<"("<<m_a<<","<<m_b<<")"<<endl;}
        Demo(int a,int b);      //Declaration
        ~Demo();
};
//Defining ctor outside body using class name and scope resolution operator
Demo::Demo(int a,int b)
{
        m_a=a;
        m_b=b;
}

Demo::~Demo()
{
        cout<<"dtor called:";
        //show called, just to demonstrate which object is going out of scope
        show();
}
int main()
{
        Demo d1(2,3);
        d1.show();
        Demo d2(4,5);
        d2.show();
}
//Notice the order of destruction of objects: reverse order of creation
/*Output:
```

```
(2,3)
(4,5)
dtor called:(4,5)
dtor called:(2,3)
*/
```

Note: The Synthesized Destructor: The compiler defines a synthesized destructor for any class that does not define its own destructor.

It is important to realize that the destructor body does not directly destroy the members themselves. Members are destroyed as part of the implicit destruction phase that follows the destructor body. A destructor body executes in addition to the memberwise destruction that takes place as part of destroying an object.

```
//classComplex9Destructor.cpp
/*Complex number using class in C++
*default constructor doesn't initialize members to default values
*if we provide our own constructor then we cannot use the constructor provided by compiler
*overload ctors: parameterized ctor with default arguments, object can be created with zero,one or two arguments
*Single constructor handles all 3 cases, because of default arguments
*automatic object creation
*user defined destructor to show the destruction order
*deleting the object gives call to destructor implicitly
*
*Memory for an object must be available before its construction and can be released only after its destruction
*Notice the destruction of local objects in reverse order of their creation
*/

#include<iostream>
using namespace std;

class Complex
{
private:
	int re;
	int im;
public:

//parameterized ctor using member initialization list and default arguments
	Complex(int r=0,int i=0):re(r),im(i)
	{
		cout<<"ctor called (re,im)="<<"("<<re<<","<<im<<")"<<endl;
	}
```

```
        void showComplex();
        ~Complex();
};

void Complex::showComplex()
{
        cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}
Complex::~Complex()
{
        cout<<"dtor called (re,im)="<<"("<<re<<","<<im<<")"<<endl;
}

int main()
{
        Complex c1;   //Calls parameterized ctor with default values in obj creation process
        c1.showComplex();
        Complex c2(2,3);      //Calls parameterized ctor in obj creation process
        c2.showComplex();

        Complex c3(4,5);      //Calls parameterized ctor in obj creation process
        c3.showComplex();
        return 0;
}
/*Output:
ctor called (re,im)=(0,0)
Complex number=0+0i
ctor called (re,im)=(2,3)
Complex number=2+3i
ctor called (re,im)=(4,5)
Complex number=4+5i
dtor called (re,im)=(4,5)
dtor called (re,im)=(2,3)
dtor called (re,im)=(0,0)
*/
```

//classComplex10ArrayOfObj.cpp

```
/*Complex number using class in C++
*Array of objects
*/

#include<iostream>
using namespace std;

class Complex
```

```
{
private:
        int re;
        int im;
public:

//parameterized ctor using member initialization list and default arguments
        Complex(int r=0,int i=0):re(r),im(i)
        {
                cout<<"ctor called (re,im)="<<"("<<re<<","<<im<<")"<<endl;
        }

        void showComplex();
        ~Complex();
};

void Complex::showComplex()
{
        cout<<"Complex number="<<re<<"+"<<im<<"i"<<endl;
}

Complex::~Complex()
{
        cout<<"dtor called (re,im)="<<"("<<re<<","<<im<<")"<<endl;
}


int main()
{
//      Complex c1[2];
        Complex c2[3]={Complex(1,1),Complex(2,2),Complex(3,3)};
        return 0;
}


/*Output:
ctor called (re,im)=(1,1)
ctor called (re,im)=(2,2)
ctor called (re,im)=(3,3)
dtor called (re,im)=(3,3)
dtor called (re,im)=(2,2)
dtor called (re,im)=(1,1)
*/
```

//classComplex11ArrOfObjGetterSetter.cpp

```
/*Complex number using class in C++
* all data members are made private and member functions are made public
* i.e., only the public member functions of the class can acess the data
* there by providing controlled access to the data
* notice the use of getter(accessor) and setter(mutator) methods
*/


#include<iostream>
using namespace std;

class Complex
{
  private:
          int re;
          int im;
  public:
          int getRE();
          int getIM();
          void setRE(int);
          void setIM(int);
          Complex();
          ~Complex();
};

Complex::Complex()
{
          cout<<"constructor called, object address:"<<this<<endl;
}
int Complex::getRE()
{
          return re;
}
int Complex::getIM()
{
          return im;
}

void Complex::setRE(int r)
{
          re=r;
}
void Complex::setIM(int i)
{
          im=i;
}
Complex::~Complex()
```

```
{
	cout<<"destructor called, object address:"<<this<<endl;
}
int main()
{
	Complex c[3];

	for(int i=0;i<3;i++)
	{
		c[i].setRE(i);
		c[i].setIM(i);
	}

	for(int i=0;i<3;i++)
	{
		cout<<"Complex
number"<<(i+1)<<":"<<c[i].getRE()<<"+"<<c[i].getIM()<<"i"<<endl;
	}
	return 0;
}

/*Output:
constructor called, object address:0x22ff00
constructor called, object address:0x22ff08
constructor called, object address:0x22ff10
Complex number1:0+0i
Complex number2:1+1i
Complex number3:2+2i
destructor called, object address:0x22ff10
destructor called, object address:0x22ff08
destructor called, object address:0x22ff00
*/
```

//globalObj.cpp

```
//global object c1
#include<iostream>

using namespace std;

class Complex
{
	double re;
	double im;

public:
	Complex(double r,double i):re(r),im(i){cout<<"ctor called: "; show();}
	void show(){cout<<re<<"+"<<im<<"i"<<endl;}
	~Complex(){cout<<"dtor called: "; show();}
};

Complex c1(2,3);
int main()
{
	cout<<"Main started"<<endl;
	Complex c2(4,5);
	cout<<"Main about to end"<<endl;

	return 0;
}

/*Output:
ctor called: 2+3i
Main started
ctor called: 4+5i
Main about to end
dtor called: 4+5i
dtor called: 2+3i
*/
```

Dynamic Memory Management using Constructors and Destructors

//dynObj.cpp

```
//Dynamic Object creation
//Object is deleted when delete is called on the pointer
//For dynamic objects:
	//new allocates memory and also calls the constructor
	//delete calls the destructor and then deallocates memory
//In dynamic allocation user has full freedom as to when to create the object by new and
//when to destroy it by delete.
//The object lifetime is limited between them(new and delete)
#include<iostream>
using namespace std;

class Complex
{
	double re;
	double im;

public:
	Complex(double r,double i):re(r),im(i){cout<<"ctor called: "; show();}
	~Complex(){cout<<"dtor called: "; show();}
	void show(){cout<<re<<"+"<<im<<"i"<<endl;}
};
```

```
int main()
{
	Complex* c1=new Complex(2,3);
	Complex* c2=new Complex(4,5);
//Dynamic objects lifetime doesn't end at the end of function
//Dynamic objects have to be deleted explicitly by calling delete on the pointer pointing to them
//Dynamic objects may be deleted in any order
	delete c1;
	cout<<"Bye c1"<<endl;

	delete c2;
	cout<<"Bye c2"<<endl;
	return 0;
}

/*Output:
ctor called: 2+3i
ctor called: 4+5i
dtor called: 2+3i
Bye c1
dtor called: 4+5i
Bye c2
*/
```

//namelessObj.cpp

```
/*Anonymous(nameless) instances are objects that you only need to construct and send to somewhere else (like calling a function), but you don't need to know anything about afterwards. Unlike other objects, anonymous instances have a scope that lasts only until the end of the expression they are created in.*/
#include<iostream>
using namespace std;

class Nameless
{
	int a;
public:
	Nameless(){cout<<"Zero arg ctor"<<endl;}
	Nameless(int a1):a(a1){cout<<"ctor called"<<endl;}
	~Nameless(){cout<<"dtor"<<endl;}
	void show(){cout<<a<<endl;}
};
int main()
{
	Nameless();
//	Nameless(2).show();
	Nameless a1(2);
```

```
        a1.show();

        return 0;
}
/*Output:
Zero arg ctor
dtor
ctor called
2
dtor
*/
```

Copy Constructor:

We know:

Complex c1 = {4.2, 5.9};   // or c1(4.2, 5.9)

invokes Constructor

Complex::Complex(double, double);

Which constructor is invoked for? Complex c2(c1); Or for? Complex c2 = c1;

It is the **Copy Constructor** that takes an object of the same type and constructs a copy. A constructor is the copy constructor if its first parameter is a reference to the class type and any additional parameters have default values. It initializes a new object as a copy of another object of the same type. The copy constructor is applied implicitly to pass objects to or from a function by value. If we do not provide the copy constructor, the compiler synthesizes one for us.

Complex::Complex(**const Complex &**);

(Synthesized or) Free Copy Constructor

If no **copy constructor** is provided by the user, the compiler supplies a **free copy constructor**.

Compiler-provided copy constructor cannot initialize the object to proper values (in all cases). It performs a shallow-copy.

Why the parameter to a copy constructor must be passed as Call-by-Reference?

MyClass(MyClass other);

The fact that the copy constructor is used to initialize nonreference parameters of class type explains why the copy constructor's own parameter must be a reference. If that parameter were not a reference, then the call would never succeed—to call the copy constructor, we'd need to use the copy constructor to copy the argument, but to copy the argument, we'd need to call the copy constructor, and so on indefinitely.

So compiler throws an error.

Signature of a Copy Constructor can be one of:

MyClass(**const MyClass&** other);   // Common, Source cannot be changed

MyClass(**MyClass&** other);   //Occasional, Source can be changed

//cctor1.cpp

```
//If the class has copy ctor, compiler doesn't provide its free default ctor.
//free copy ctor does shallow copy
//Here user has defined his own copy ctor to demonstrate copy ctor's general syntax(in
//function header).
//This pgm dactually oesnt need the user defined copy ctor as the object doesn't hold any
//resources outside

#include<iostream>
using namespace std;

class Complex
{
	double re;
	double im;

public:
	Complex(double r,double i):re(r),im(i){cout<<"ctor called "; show();}
	Complex(const Complex& c):re(c.re),im(c.im){cout<<"cctor called "; show();}
	~Complex(){cout<<"dtor called:"; show();}
	void show(){cout<<re<<"+"<<im<<"i"<<endl;}
};
int main()
{
	Complex c1(2,3);
	Complex c2(c1); c2.show();
	return 0;
}
/*Output:
ctor called 2+3i
cctor called 2+3i
2+3i
dtor called:2+3i
dtor called:2+3i
*/
```

Why do we need Copy Constructor?

Consider the function call mechanisms in C++:

Call-by-reference: Set a reference to the actual parameter as a formal parameter. Both the formal parameter and the actual parameter share the same location (object)

Return-by-reference: Set a reference to the computed value as a return value. Both the computed value and the return value share the same location (object)

Call-by-value: Make a copy (clone) of the actual parameter as a formal parameter. This needs a Copy Constructor

Return-by-value: Make a copy (clone) of the computed value as a return value. This needs a Copy Constructor

Copy Constructor is needed for initializing the data members of a UDT from an existing value.

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When the compiler generates a temporary object.

It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example is the return value optimization (sometimes referred to as RVO). (Optional Reading link) https://en.wikipedia.org/wiki/Copy_elision#Return_value_optimization

//cctor2CallbyVal.cpp

```
//Notice the call to copy ctor when the function showMe passes an object by
//callByValue
//Here I have made showMe as a non member function of class. So it uses
//show(member function) to access private members

#include<iostream>
using namespace std;

class Complex
{
        double re;
        double im;

public:
        Complex(double r,double i):re(r),im(i){cout<<"ctor called "; show();}
        Complex(const Complex& c):re(c.re),im(c.im){cout<<"cctor called "; show();}
        ~Complex(){cout<<"dtor called:"; show();}
        void show(){cout<<re<<"+"<<im<<"i"<<endl;}
};

void showMe(Complex c){cout<<"I'm in showMe"; c.show();}
int main()
{
        Complex c1(2,3);
```

```
	Complex c2(c1);
	showMe(c2);
	return 0;
}
/*Output:
ctor called 2+3i
cctor called 2+3i
cctor called 2+3i
I'm in showMe2+3i
dtor called:2+3i
dtor called:2+3i
dtor called:2+3i
*/
```

//cctor3RetByVal.cpp

```
//Notice the call to copy ctor when the member function showMe returns an object by Value
//Here (on g++) copy Constructor has to take constant reference const Complex&
//Because object is returned by Value
#include<iostream>

using namespace std;

class Complex
{
	double re;
	double im;

public:
	Complex(double r,double i):re(r),im(i){cout<<"ctor called "; show();}
	Complex(const Complex& c):re(c.re),im(c.im){cout<<"cctor called "; show();}
//Removing const in copy ctor parameter is a compilation error on g++, because c2.showMe() //is returning object by value
	~Complex(){cout<<"dtor called:"; show();}
	void show(){cout<<re<<"+"<<im<<"i"<<endl;}
	Complex showMe(){cout<<"I'm in showMe: "; return *this;}
};

int main()
{
	Complex c1(2,3);
	Complex c2(c1);
```

```
        Complex c3=c2.showMe();
        return 0;
}

/*Output:
ctor called 2+3i
cctor called 2+3i
I'm in showMe: cctor called 2+3i
dtor called:2+3i
dtor called:2+3i
dtor called:2+3i
*/
```

The Rule of Three/Five

There are three basic operations to control copies of class objects:
the copy constructor, copy-assignment operator, and destructor.
Under the new standard, a class can also define a move constructor and move-assignment operator.

Classes That Need Destructors Need Copy and Assignment.

One rule of thumb to use when you decide whether a class needs to define its own versions of the copy-control members is to decide first whether the class needs a destructor.
Often, the need for a destructor is more obvious than the need for the copy constructor or assignment operator. If the class needs a destructor, it almost surely needs a copy constructor and copy-assignment operator as well.

//cctor4aNeed.cpp

```
//Run time error or incorrect output
//double free
//This pgm clearly demonstrates the need of copy ctor to perform deep copying
//Because compiler's free copy ctor will be called for c2(c1). Free copy ctor does shallow copy

//See cctor4bNeed.cpp where in user has defined the copy ctor to perform deep copy
#include<iostream>
#include<cstring>
using namespace std;
class CctorNeed
{
        char *str;
        int len;
public:
        CctorNeed(char *s)
        {
                str=new char[strlen(s)+1];
                strcpy(str,s);
```

```
			len=strlen(str);
			cout<<"ctor called "<<this<<endl;  //this gives the current objects address. You may comment out
			show();
		}
		~CctorNeed()
		{
			cout<<"dtor called "<<this<<endl;show();//this gives the current objects address. You may comment out
			delete []str;
		}

		void show(){cout<<"name="<<str<<" Length="<<len<<endl;}
};
int main()
{
		CctorNeed c1("C++");
		CctorNeed c2(c1);
		return 0;
}
```

//cctor4bNeed.cpp

```
//Same as cctor4aNeed.cpp, except that here user has defined own copy ctor as per  the
//class requirement
//Deep copy takes place. So we get proper output.
#include<iostream>
#include<cstring>
using namespace std;

class CctorNeed
{
		char *str;
		int len;
public:
		CctorNeed(char *s)
		{
			str=new char[strlen(s)+1];
			strcpy(str,s);
			len=strlen(str);
		cout<<"ctor called "<<this<<endl;     //this gives the current objects adrss. You may comment
out
			show();
		}
		CctorNeed(const CctorNeed& s)
		{
			str=new char[s.len+1];
```

```
                strcpy(str,s.str);
                len=strlen(str);
        cout<<"cctor called "<<this<<endl;//this gives  current object addrs. You may comment out
                show();
        }

        ~CctorNeed()
        {
cout<<"dtor called "<<this<<endl;show();  //this gives  current objects address.You may comment out
                delete []str;
        }
        void show(){cout<<"name="<<str<<" Length="<<len<<endl;}
};
int main()
{
        CctorNeed c1("C++");
        CctorNeed c2(c1);
        return 0;
}
/*Output:
ctor called 0x7ffcf69b7040
name=C++ Length=3
cctor called 0x7ffcf69b7050
name=C++ Length=3
dtor called 0x7ffcf69b7050
name=C++ Length=3
dtor called 0x7ffcf69b7040
name=C++ Length=3
*/
```

The copy-assignment operator:

- Just as a class controls how objects of that class are initialized, it also controls how objects of its class are assigned using Copy-assignment operator. It is the version of the assignment operator that takes an object of the same type as its type.
- Ordinarily, the copy-assignment operator has a parameter that is a reference to const and returns a reference to its object. The compiler synthesizes the copy-assignment operator if the class does not explicitly provide one.

Overloaded operators are functions that have the name operator followed by the symbol for the operator being defined. Hence, the assignment operator is a function named operator=. Like any other function, an operator function has a return type and a parameter list.

The parameters in an overloaded operator represent the operands of the operator. Some operators, assignment among them, must be defined as member functions. When an operator is a member function, the left-hand operand is bound to the implicit this parameter. The right-hand operand in a binary operator, such as assignment, is passed as an explicit parameter.

The copy-assignment operator takes an argument of the same type as the class:

```
class Foo {
public:
Foo& operator=(const Foo&); // assignment operator
// ...
};
```

To be consistent with assignment for the built-in types, assignment operators usually return a reference to their left-hand operand (allows chaining of assignment operator. Eg: a=b=c where a,b,c are objects).

//cpyAssign1.cpp

```
//If the class has copy assignment function, compiler doesn't provide its free copy
//assignment  function
//Compiler defined copy assignment function does shallow copying
//This pgm demonstrates the syntax(see operator= function header) of overloading
//assignment operator(=)
//This pgm actually doesnt need the user defined copy assignment operator as the object
//doesn't  hold any resources outside
#include<iostream>
using namespace std;

class Complex
{
	double re;
	double im;

public:
	Complex(double r,double i):re(r),im(i){cout<<"ctor called "; show();}
	Complex(const Complex& c):re(c.re),im(c.im){cout<<"cctor called "; show();}
	//Overloading of assignment operator
	Complex& operator=(const Complex& c)
	{
		re=c.re;
		im=c.im;
		cout<<"copy assign called ";
		show();
		return *this;
	}
```

```
        ~Complex(){cout<<"dtor called:"; show();}
        void show(){cout<<re<<"+"<<im<<"i"<<endl;}
};
int main()
{
        Complex c1(2,3);        //Param ctor called
        Complex c2(c1);         //Copy ctor called
        Complex c3(4,5);        //Param ctor called
        c3=c1;                  //overloaded function i.e., copy assignment is called
        return 0;
}
/*Output:
ctor called 2+3i
cctor called 2+3i
ctor called 4+5i
copy assign called 2+3i
dtor called:2+3i
dtor called:2+3i
dtor called:2+3i
*/
```

//cpyAssign2a.cpp

```
//This program doesn't give proper output. It demonstrates the need of user defind copy
//assignment function
//Compiler's free copy assignment function does shallow copying
//See cpyAssign2b.cpp
#include<iostream>
#include<cstring>
using namespace std;

class String
{
        char *str;
        int len;
public:
        String(char *s)
        {
                str=new char[strlen(s)+1];
                strcpy(str,s);
                len=strlen(str);
                cout<<"ctor called "<<endl;
                show();
        }
```

```
	String(const String& s)
	{
		str=new char[s.len+1];
		strcpy(str,s.str);
		len=strlen(str);
		cout<<"cctor called "<<endl;
		show();
	}

	~String()
	{
		cout<<"dtor called "<<endl;show();
		delete []str;
	}

	void show(){cout<<"name="<<str<<" Length="<<len<<endl;}
};
int main()
{
	String c1("C++");
	String c2=c1;			//copy ctor called
	String c3("Pgm");
	c3=c1;				//copy assignment called
			//here user has not defined copy assignment, so compiler's free copy
			//assignment will be called
	return 0;
}
//Wrong output because of shallow copy by synthesized assignment operator
/*Output:
ctor called
name=C++ Length=3
cctor called
name=C++ Length=3
ctor called
name=Pgm Length=3
dtor called
name=C++ Length=3
dtor called
name=C++ Length=3
dtor called
name= Length=3
*/
```

```
//cpyAssign2b.cpp
//Here user has defined own copy assignment function, so deep copy happens and we get
//proper output
#include<iostream>
#include<cstring>

using namespace std;

class String
{
	char *str;
	int len;
public:
	String(char *s)
	{
		str=new char[strlen(s)+1];
		strcpy(str,s);
		len=strlen(str);
		cout<<"ctor called "<<endl;
		show();
	}

	String(const String& s)
	{
		str=new char[s.len+1];
		strcpy(str,s.str);
		len=strlen(str);
		cout<<"cctor called "<<endl;
		show();
	}

	~String()
	{
		cout<<"dtor called "<<endl;show();
		delete []str;
	}

//User defined copy assignment function
	String& operator=(const String& s)
	{
		delete []str;                    // Release existing memory
		str=new char[s.len+1];           // Perform deep copy
```

```
                strcpy(str,s.str);
                len=s.len;
                cout<<"copy assign called "<<endl;
                return *this;                   // Return object for chain assignment
        }

        void show(){cout<<"name="<<str<<" Length="<<len<<endl;}
};
int main()
{
        String c1("C++");
        String c2(c1);
        String c3("Pgm");
        c3=c1;
        return 0;
}
/*Output:
ctor called
name=C++ Length=3
cctor called
name=C++ Length=3
ctor called
name=Pgm Length=3
copy assign called
dtor called
name=C++ Length=3
dtor called
name=C++ Length=3
dtor called
name=C++ Length=3
*/
```

//selfCpyAssignPrblm.cpp

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

class String
{
        char *str;
        int len;
public:
        String(char *s)
        {
                str=new char[strlen(s)+1];
                strcpy(str,s);
                len=strlen(str);
                cout<<"ctor called "<<endl;
                show();
        }
        String(const String& s)
        {
                str=new char[s.len+1];
                strcpy(str,s.str);
                len=s.len;
                cout<<"cctor called "<<endl;
                show();
        }

        ~String()
        {
        //      cout<<"dtor called "<<endl;show();
                delete []str;
        }
        String& operator=(const String& s)
        {
                cout<<"copy assign called "<<endl;

                delete []str;                           // Release existing memory
                str=new char[s.len+1];                  // Perform deep copy
                strcpy(str,s.str);
                len=s.len;
                return *this;                       // Return object for chain assignment
        }
        void show(){cout<<"name="<<str<<" Length="<<len<<endl;}
};
int main()
{
        String c1("C++");
        String c2(c1);
        String c3("Pgms");
        c1=c3;
        c1.show();
        c1=c1;     //we do not get proper output, because self copy assignment not handled
        c1.show();
        return 0;
```

```
}
//Self copy assignment not handled, so wrong output
/*Output:
ctor called
name=C++ Length=3
cctor called
name=C++ Length=3
ctor called
name=Pgms Length=4
copy assign called
name=Pgms Length=4
copy assign called
name= Length=4
*/
```

```
//selfCpyAssignSolved.cpp
//This program solves the self copy assignment problem
//This should be the syntax used for copy assignment
#include<iostream>
#include<cstring>
using namespace std;
class String
{
        char *str;
        int len;
public:
        String(char *s)
        {
                str=new char[strlen(s)+1];
                strcpy(str,s);
                len=strlen(str);
                cout<<"ctor called "<<endl;
                show();
        }
        String(const String& s)
        {
                str=new char[s.len+1];
                strcpy(str,s.str);
                len=strlen(str);
                cout<<"cctor called "<<endl;
                show();
        }
        ~String()
        {
//              cout<<"dtor called "<<this<<endl;show();
                delete []str;
        }

        String& operator=(const String& s)
        {
                cout<<"copy assign called "<<endl;
                if(this!=&s) {                          //handles self assignment
                  delete []str;                         // Release existing memory
                  str=new char[strlen(s.str)+1];        // Perform deep copy
                  strcpy(str,s.str);
                  len=s.len;
                }
                return *this;                   // Return object for chain assignment
        }
        void show(){cout<<"name="<<str<<" Length="<<len<<endl;}
};
int main() {
        String c1("C++");               //ctor
        String c2(c1);                  //copy ctor
        String c3("Pgms");              //ctor
        c3.show();
        c1=c1;                          //copy assignment, calls overloaded operator= function
        c1.show();
        return 0;
}
/*Output:
ctor called
name=C++ Length=3
cctor called
name=C++ Length=3
ctor called
name=Pgms Length=4
name=Pgms Length=4
copy assign called
name=C++ Length=3
*/
```

For class **MyClass**, typical copy assignment operator will be:

```
MyClass& operator=(const MyClass& s) {
if (this != &s) {
// Release resources held by *this
// Copy members of s to members of *this
}
return *this;
}
```

Rvalue References:

In C++, an lvalue is something of which you can take an address; a named variable, for example. The name comes from the fact that they normally appear on the left-hand side of an assignment. An rvalue on the other hand is anything that is not an lvalue such as a constant value, or a temporary object or value. Typically an rvalue is on the right-hand side of an assignment operator.
An rvalue reference is a reference to an rvalue. In particular, it is a concept that is applied when the rvalue is a temporary object. The purpose of an rvalue reference is to make it possible for a particular function to be chosen when a temporary object is involved. The consequence of this is that certain operations that normally involve copying large values can be implemented by copying pointers to those values, knowing the temporary object will be destroyed.

A function can specify an rvalue reference parameter by using && as part of the parameter specification; e.g., type&& name . Normally, a temporary object will be seen as a const type& , but when there is a function overload that uses an rvalue reference, a temporary object can be resolved to that overload. The following example demonstrates this. The code first defines two incr() functions, one accepting an lvalue reference and one accepting an rvalue reference.

```
// Increment value using lvalue reference parameter.
void incr(int& value)
{
cout <<"increment with lvalue reference"<< endl;
++value;
}
// Increment value using rvalue reference parameter.
void incr(int&& value)
{
cout <<"increment with rvalue reference"<< endl;
++value;
}
```

You can call the incr() function with a named variable as argument. Because a is a named variable, the incr() function accepting an lvalue reference is called. After the call to incr() , the value of a will be 11.

```
int a = 10, b = 20;
incr(a);            // Will call incr(int& value)
```

You can also call the incr() function with an expression as argument. The incr() function accepting an lvalue reference cannot be used, because the expression a + b results in a temporary, which is not an lvalue. In this case the rvalue reference version is called. Since the argument is a temporary, the incremented value is lost after the call to incr() .

```
incr(a + b);     // Will call incr(int&& value)
```

A literal can also be used as argument to the incr() call. This will also trigger a call to the rvalue reference version because a literal cannot be an lvalue.

```
incr(3);         // Will call incr(int&& value)
```

Note: Rvalue is a C++11 feature, before c++11, to call incr(3) or incr(a+b) the function header of **incr** should have had a const qualifier, like, void incr(const int& value). Otherwise these function calls would turn out to be an error.

If you remove the incr() function accepting an lvalue reference, calling incr() with a named variable like incr(b) will result in a compiler error because an rvalue reference parameter (int&& value) will never be bound to an lvalue (b). You can force the compiler to call the rvalue reference version of incr() by using **std::move()**, which converts an lvalue into an rvalue as follows. After the incr() call, the value of b will be 21.

```
incr(std::move(b));     // Will call incr(int&& value)
```

Rvalue references are not limited to parameters of functions. You can declare a variable of type rvalue reference, and assign to it, although this usage is uncommon. Consider the following code, which is illegal in C++:

```
int& i = 2;             // Invalid: reference to a constant
int a = 2, b = 3;
int& j = a + b;         // Invalid: reference to a temporary
```

Using rvalue references, the following is perfectly legal:

```
int&& i = 2;
int a = 2, b = 3;
int&& j = a + b;
```

Stand-alone rvalue references, as in the preceding example, are rarely used as such.

```
//moveBasics.cpp
#include<iostream>
using namespace std;
int main() {
        int a=10;
        int &b=a;               //b is an lvalue reference
        cout<<a<<""<<b<<endl;

        int&& e=move(a);        //e is an rvalue reference, move converts the lvalue a to rvalue
        int&& f=20;             //f is an rvalue reference
        cout<<e<<""<<f<<endl;
        return 0;
}
//g++ -std=c++11 pgm32_moveBasics.cpp
/*Output:
```

```
10 10
10 20
*/
```

Move Semantics:

Move semantics for objects requires a **move constructor** and a **move assignment operator**. These will be used by the compiler on places where the source object is a temporary object that will be destroyed after the copy or assignment. Both the move constructor and the move assignment operator copy/move the member variables from the source object to the new object and then reset the variables of the source object to null values. By doing this, they are actually moving ownership of the memory from one object to another object. They basically do a shallow copy of the member variables and switch ownership of allocated memory to prevent dangling pointers or memory leaks.

Move semantics is implemented by using rvalue references. To add move semantics to a class, a move constructor and a move assignment operator need to be implemented. Move constructors and move assignment operators should be marked with the noexcept qualifier to tell the compiler that they don't throw any exceptions. This is particularly important for compatibility with the standard library, as fully compliant implementations of the standard library will only move stored objects if, having move semantics implemented, they also guarantee not to throw.

move Library function: used to bind an rvalue reference to an lvalue. Calling move implicitly promises that we will not use the moved-from object except to destroy it or assign a new value to it.

move constructor: Constructor that takes an rvalue reference to its type. Typically, a move constructor moves data from its parameter into the newly created object. After the move, it must be safe to run the destructor on the given argument.

move-assignment operator: Version of the assignment operator that takes an rvalue reference to its type. Typically, a move-assignment operator moves data from the right-hand operand to the left. After the assignment, it must be safe to run the destructor on the right-hand operand.

//moveSemantics.cpp

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

class Holder
{
	int *m_data;
	int m_size;

public: Holder(int size)			//Parameterized ctor
	{
        cout<<"Param ctor called: "<<this<<endl;
        m_data=new int[size];
        m_size=size;
        for(auto i=0;i<m_size;i++)
                m_data[i]=i;
        cout<<"show from param ctor: ";show();
}

~Holder()                                       //Destructor
{
        cout<<"dtor called: "<<this<<endl;
        delete[] m_data;
}

Holder(const Holder& other)                     //Copy ctor
{
        cout<<"Copy ctor called"<<endl;
        cout<<"I'm: "<<this<<endl;
        m_data=new int[other.m_size];
        copy(other.m_data,other.m_data+other.m_size,m_data);
        m_size=other.m_size;
        cout<<"show from copy ctor: ";show();
}

Holder& operator=(const Holder& other)                  //Copy assignment operator
{
        cout<<"Copy assignment called"<<endl;
        if(this!=&other)
        {
                delete[] m_data;
                m_data=new int[other.m_size];
                copy(other.m_data,other.m_data+other.m_size,m_data);
                m_size=other.m_size;
        }
        cout<<"show from assign overload: ";show();
        return *this;
}
Holder(Holder&& other)                          //Move copy ctor
{
        cout<<"Mov ctor called"<<endl;
        cout<<"I'm: "<<this<<endl;
        cout<<"Before Moving, other contains: ";
        for(auto i=0;i<other.m_size;i++)
```

```
                cout<<other.m_data[i]<<"";
        cout<<endl;
        m_data=other.m_data;                    //Shallow copy
        m_size=other.m_size;

        cout<<"Moved"<<endl;
        other.m_data=nullptr;           //Make the state of the moved object safe
        other.m_size=0;
        cout<<"new m_data contains: ";show();
        cout<<"After Moving, other contains: ";
        if(other.m_data==nullptr)
                cout<<"nothing";
        else
                for(auto i=0;i<other.m_size;i++)
                        cout<<other.m_data[i]<<"";
        cout<<endl;
}
Holder& operator=(Holder&& other)//Move assignment operator
{
        cout<<"Move assignment called"<<endl;
        cout<<"Before Move assigning, other contains: ";
        for(auto i=0;i<other.m_size;i++)
                cout<<other.m_data[i]<<"";
        cout<<endl;
        if(this!=&other)                //To avoid self assignment problem
        {
                delete[] m_data;        //To avoid memory leak
                m_data=other.m_data;
                m_size=other.m_size;

                cout<<"Moved"<<endl; //Make the state of the moved object safe
                other.m_data=nullptr;
                other.m_size=0;
                cout<<"new m_data contains: ";show();
                cout<<"After Move assigning, other contains: ";
                if(other.m_data==nullptr)
                cout<<"nothing";
                else
                        for(auto i=0;i<other.m_size;i++)
                                cout<<other.m_data[i]<<"";
                cout<<endl;
        }
        return *this;
```

```
		}
		void show()
		{
			for(auto i=0;i<m_size;i++)
				cout<<m_data[i]<<"";
			cout<<endl;
		}
};

Holder createHolder(int size)				//Function to create temporary object
{
		cout<<"returning from create holder"<<endl;
		return Holder(size);
}

int main()
{
		Holder h1=move(createHolder(5));
		Holder h2(h1);
		h2=createHolder(10);
		h2=h1;
		return 0;
}

//g++ -std=c++11 pgm33_moveSemantics.cpp
/*Output:
returning from create holder
Param ctor called: 0x7fff1b60e2e0
show from param ctor: 0 1 2 3 4
Mov ctor called
I'm: 0x7fff1b60e2c0
Before Moving, other contains: 0 1 2 3 4
Moved
new m_data contains: 0 1 2 3 4
After Moving, other contains: nothing
dtor called: 0x7fff1b60e2e0
Copy ctor called
I'm: 0x7fff1b60e2d0
show from copy ctor: 0 1 2 3 4
returning from create holder
Param ctor called: 0x7fff1b60e2e0
show from param ctor: 0 1 2 3 4 5 6 7 8 9
Move assignment called
```

```
Before Move assigning, other contains: 0 1 2 3 4 5 6 7 8 9
Moved
new m_data contains: 0 1 2 3 4 5 6 7 8 9
After Move assigning, other contains: nothing
dtor called: 0x7fff1b60e2e0
Copy assignment called
show from assign overload: 0 1 2 3 4
dtor called: 0x7fff1b60e2d0
dtor called: 0x7fff1b60e2c0
*/
```

//Move semantics: without many messages in functions (as in above pgm)

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

class Holder
{
        int *m_data;
        int m_size;

        public:
                Holder(int size)
                {
                        cout<<"ctor cald"<<endl;
                        m_data=new int[size];
                        for(int i=0;i<size;i++)
                        {
                                m_data[i]=i;
                        }
                        m_size=size;
                }

                ~Holder()
                {
                        cout<<"dtror cald"<<endl;
                        delete []m_data;
                }

                void display()
                {
                        cout<<"Data:";
                        for(int i=0;i<m_size;i++)
                        {
```

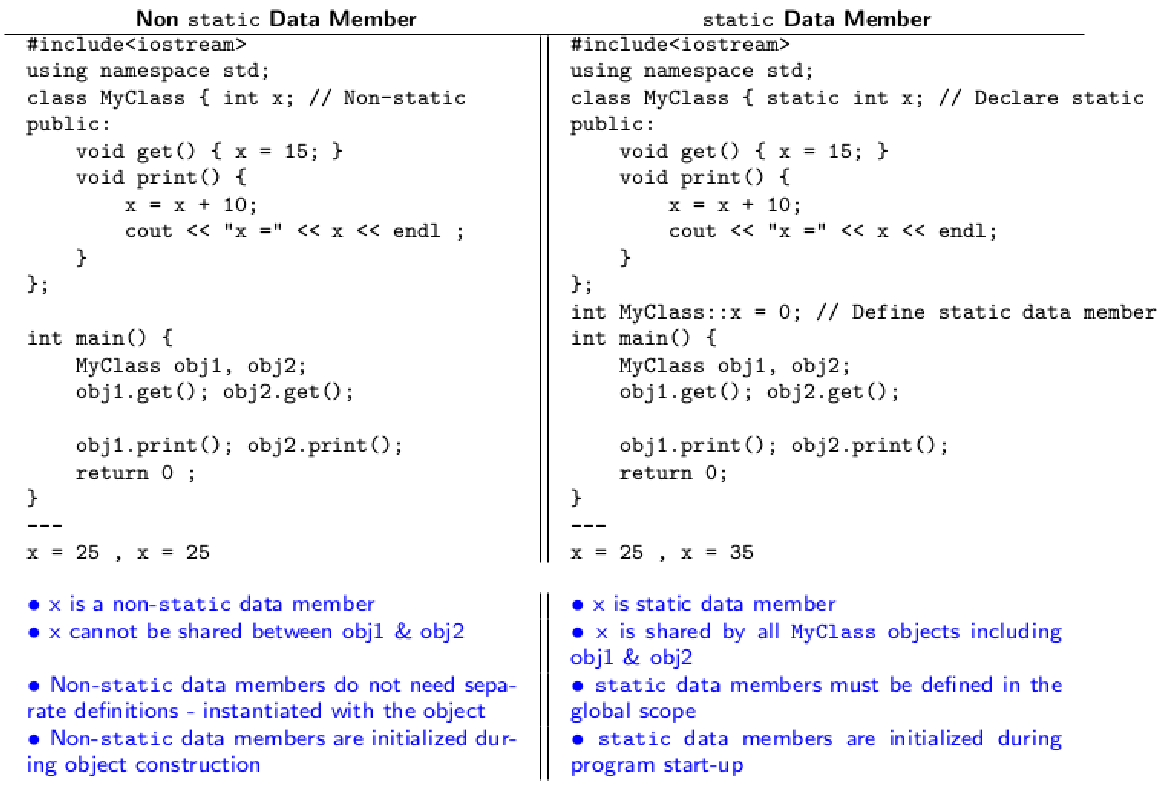
```
                        cout<<m_data[i]<<" ";
                }
                cout<<"size="<<m_size<<endl;
        }

//Copy ctor
        Holder(const Holder& other)
        {
                cout<<"cctor cald"<<endl;
                m_data=new int[other.m_size];
                copy(other.m_data,other.m_data+other.m_size,m_data);

/*
//OR for loop for copying
                        for(int i=0;i<other.m_size;i++)
                        {
                                m_data[i]=other.m_data[i];
                        }
*/
                m_size=other.m_size;
        }

//Move copy ctor
        Holder(Holder&& other)
        {
                cout<<"Move cctor cald"<<endl;

//Shallow copy, Steal resources from temporary obj
                m_data=other.m_data;
                m_size=other.m_size;

//Make the state of temporary object safe
                other.m_data=nullptr;
                other.m_size=0;
        }

//Copy assignment operator overloaded
    Holder& operator=(const Holder& other)
    {
        cout<<"Copy assignment called"<<endl;
        if(this!=&other)
        {
                delete[] m_data;
                m_data=new int[other.m_size];
                copy(other.m_data,other.m_data+other.m_size,m_data);
                m_size=other.m_size;
            }
            return *this;
        }

//Move assignment operator overloaded
            Holder& operator=(Holder&& other)
            {
                cout<<"Move assignment called"<<endl;
                if(this!=&other)                //Handle self assignment
                {
                    delete[] m_data;            //To avoid memory leak

//Shallow copy,Steal resources from temporary obj

                    m_data=other.m_data;
                    m_size=other.m_size;

//Make the state of temporary object safe
                    other.m_data=nullptr;
                    other.m_size=0;
                }
                return *this;
            }
};

Holder createHolder(int size)
{
        return Holder(size);
}
int main()
{
        Holder h1(5);
        h1.display();

        Holder h2(h1);                  //cctor
        h2.display();

        Holder h3=move(createHolder(10));       //move cctor
        h3.display();
```

```
	h3=h1;                                  //copy assignment operator
	h3.display();

	h3=move(createHolder(5));               //move assignment operator
	h3.display();

	return 0;
}

/*
Output:
ctor cald
Data:0 1 2 3 4 size=5
cctor cald
Data:0 1 2 3 4 size=5
ctor cald
Move cctor cald
dtror cald
Data:0 1 2 3 4 5 6 7 8 9 size=10
Copy assignment called
Data:0 1 2 3 4 size=5
ctor cald
Move assignment called
dtror cald
Data:0 1 2 3 4 size=5
dtror cald
dtror cald
dtror cald
*/
```

Static Members

Static Data Member

- is associated with **class** not with **object**
- is shared by all the objects of a class
- needs to be defined outside the class scope (in addition to the declaration within the class scope) to avoid linker error
- must be initialized in a source file
- can be private / public type
    - can be accessed with the class-name followed by the scope resolution operator (::)
    - as a member of any object of the class



Non static Data Member

```
#include<iostream>
using namespace std;
class MyClass { int x; // Non-static
public:
    void get() { x = 15; }
    void print() {
        x = x + 10;
        cout << "x =" << x << endl ;
    }
};

int main() {
    MyClass obj1, obj2;
    obj1.get(); obj2.get();

    obj1.print(); obj2.print();
    return 0 ;
}
---
x = 25 , x = 25
```

- x is a non-static data member
- x cannot be shared between obj1 & obj2

- Non-static data members do not need separate definitions - instantiated with the object
- Non-static data members are initialized during object construction

static Data Member

```
#include<iostream>
using namespace std;
class MyClass { static int x; // Declare static
public:
    void get() { x = 15; }
    void print() {
        x = x + 10;
        cout << "x =" << x << endl;
    }
};
int MyClass::x = 0; // Define static data member
int main() {
    MyClass obj1, obj2;
    obj1.get(); obj2.get();

    obj1.print(); obj2.print();
    return 0;
}
---
x = 25 , x = 35
```

- x is static data member
- x is shared by all MyClass objects including obj1 & obj2
- static data members must be defined in the global scope
- static data members are initialized during program start-up

Static Member Function

A static member function

- does not have this pointer – not associated with any object
- cannot access non-static data members
- cannot invoke non-static member functions
- can be accessed
  - with the class-name followed by the scope resolution operator (::)
  - as a member of any object of the class
- is needed to read / write static data members
- Again, for encapsulation static data members should be private. get()-set() idiom is built for access (static member functions in public)
- may initialize static data members even before any object creation
- cannot co-exist with a non-static version of the same function
- cannot be declared as const

//staticDataMember.cpp
//static data members are declared inside class and defined outside the class.
//non static member functions can access both static and non static data

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

class staticDemo
```

```
{
        private: static int a;              //Declaration

        public: static int b;               //Declaration
                int getA(){return a;}
};
int staticDemo::a=10;           //Definition
int staticDemo::b=20;           // Definition
int main()
{
  cout<<"static b="<<staticDemo::b<<endl;           //Okay, because b is public

  //Below line is an error, because a is private
  //      cout<<"Static a="<<staticDemo::a<<endl;

  staticDemo obj1,obj2;
  cout<<"static a="<<obj1.getA()<<endl;   //Okay, because private a accessed via member fn
  cout<<"static a="<<obj2.getA()<<endl;   //Okay, because private a accessed via member fn

//But static data can be accessed without object creation, therefore use static member function
//to access static data as shown in staticMemberFn.cpp program
        return 0;
}
/*Output:
static b=20
static a=10
static a=10
*/
```

//staticMemberFn.cpp

```
//static data members are declared(with keyword static) inside class
//and should be defined(without keyword static) outside the class.
//static member functions can access only static data
//non static member functions can access both static and non static data
#include<iostream>
using namespace std;
class staticDemo
{
        private: static int a;              //Declaration
                static int b;               //Declaration

        public:
                static int getA(){return a;}
                static int getB(){return b;}
                static void changeData(){a++;b++;}
};
int staticDemo::a=0;                        //Definition
int staticDemo::b=0;                          //Definition
int main()
{
  cout<<"static a="<<staticDemo::getA()<<endl; //Okay, because private a accessed via member fn
  cout<<"static b="<<staticDemo::getB()<<endl; //Okay, because private a accessed via member fn

  staticDemo obj1,obj2;
  staticDemo::changeData();
  cout<<"static a="<<staticDemo::getA()<<endl; //static fn accessed by class name followed by ::
  cout<<"static b="<<staticDemo::getB()<<endl; //accessed by class name followed by ::
  obj1.changeData();                                    //static fn accessed by object
  cout<<"static a="<<obj1.getA()<<endl;          //static fn accessed by object
  cout<<"static b="<<obj1.getB()<<endl;          //static fn accessed by object
  cout<<"static a="<<obj2.getA()<<endl;          //static fn accessed by object
  cout<<"static b="<<obj2.getB()<<endl;
  return 0;
}
/*Above program also demonstrates, static member fn may be accessed by classname or object*/
/*Output:
static a=0
static b=0
static a=1
static b=1
static a=2
static b=2
static a=2
static b=2
*/

//stud.cpp
//static Data Member belongs to class, not to object
//Only one copy of it is maintained, which may be accessed by classname followed by::
//or by any object
#include<iostream>
using namespace std;

class Student
{
        int m_id;
        string m_name;

  public:
         static int noOfStud;                    //public static data member
         Student(int id,string name)             //ctor
        {
                m_id=id;
                m_name=name;
                noOfStud++;
        }
        void disp()
        {
                cout<<"id="<<m_id<<" name="<<m_name<<endl;
        }
};
int Student:: noOfStud=0;

int main()
{
        Student s1(1,"abc");
        Student s2(2,"def");

        //static Data Member accessed by classname followed by ::
        cout<<"Number of Students="<<Student::noOfStud<<endl;

        //static Data Member accessed by object
        cout<<"Number of Students="<<s1.noOfStud<<endl;
        cout<<"Number of Students="<<s2.noOfStud<<endl;

        cout<<"Student 1:";
        s1.disp();
        cout<<"Student 2:";
        s2.disp();
        return 0;
}
/*Output:
Number of Students=2
Number of Students=2
Number of Students=2
Student 1:id=1 name=abc
Student 2:id=2 name=def
*/
```

One more example program to demonstrate static data member and static member functions

```
//printJobs.cpp
//when defining static member function outside the class do not use static again
#include<iostream>
using namespace std;
class PrintJobs {
        int m_pages;                    // # of pages in current job
        static int m_trayPages;  // # of pages remaining in the tray
        static int m_nJobs;             // # of print jobs executing

  public:
        PrintJobs(int pages)            //ctor
```

```
	{
		m_pages=pages;
		m_nJobs++;
		m_trayPages-=pages;
		cout<<"Printing "<<m_pages<<" pages ..."<<endl;
	}
	~PrintJobs()	{m_nJobs--; }

	static int getJobs()				//defining static member fn inside class
	{
		return m_nJobs;
	}

	static int getTrayPages();			//declaring static member fn inside class

	static void loadPages(int pages)		//defining static member fn inside class
	{
		m_trayPages+=pages;
	}

};
int PrintJobs::m_trayPages=500;			//Printer has 500 pages loaded
int PrintJobs::m_nJobs=0;			//Initially, no jobs

int PrintJobs::getTrayPages()			//defining static member fn outside class
{
	return m_trayPages;
}

int main()
{
	cout<<"Jobs="<<PrintJobs::getJobs()<<endl;
	cout<<"Pages in tray="<<PrintJobs::getTrayPages()<<endl<<endl;
	{
		PrintJobs j1(100);
		cout<<"Jobs="<<PrintJobs::getJobs()<<endl;
		cout<<"Job1 completed"<<endl<<endl;
	}
	cout<<"Pages in tray="<<PrintJobs::getTrayPages()<<endl;
	cout<<"Jobs="<<PrintJobs::getJobs()<<endl;
	cout<<"Loading 50 pages..."<<endl;
	PrintJobs::loadPages(50);
	cout<<"After loading, Pages in tray="<<PrintJobs::getTrayPages()<<endl;
	return 0;
}
/*Output:
Jobs=0
```

```
Pages in tray=500

Printing 100 pages ...
Jobs=1
Job1 completed

Pages in tray=400
Jobs=0
Loading 50 pages...
After loading, Pages in tray=450
*/
```

******************************UNIT 2 Complete******************************

Reference used for MoveSemantics:

https://www.internalpointers.com/post/c-rvalue-references-and-move-semantics-beginners