CS-1004 Object Oriented programming
Week 4

Instructor:
ATIYA
LECTURER
(COMPUTER SCIENCE DEPARTMENT)
NATIONAL UNIVERSITY- FAST (KHI CAMPUS)
EMAIL: ATIYA.JOKHIO@NU.EDU.PK
ROOM: (LECTURER ROOM#19, CS DEPT BASEMENT-2)

Copy Constructor

- ✓Creating a copy of an object means to create an exact replica of the object having the same literal value, data type, and resources.
- ✓A **copy constructor** is a type of constructor that creates an object using another object of the same class. The process of initializing members of an object through a copy constructor is known as **copy initialization**.
- ✓It is also called **member-wise initialization** because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.

Initializing one object with another

```
class A{
 int val;
 public:
 A(int val1) {
    val = val1; }
 A(){ }
 void setVal(int val1) { val = val1; }
 void showVal() { cout << "Value: "  << val  << endl; }
};
```

Initializing one object with another

```
int main(){
  A  a1( 10 );
  a1.showVal( );

  A a2  =  a1;
  a2.showVal( );
  a2.setVal( 20 );

  a1.showVal( );
  a2.showVal( );
}
```

Initializing one object with another

```
int main()
{
  A  a1( 10 );
  a1.showVal( );

  A a2  =  a1; //Copy Constructor
  a2.showVal( );
  a2.setVal( 20 );

  a1.showVal( );
  a2.showVal( );
}
```

OUTPUT:

Val: 10 *a1.val*
Val: 10 *a2.val*

Val: 10 *a1.val*
Val: 20 *a2.val*

Initializing one object with another

```
int main()
{
  A  a1( 10 );
  a1.showVal( );

  A  a2;
  a2  =  a1; // implicit Copy assignment operator
  a2.showVal( );
  a2.setVal( 20 );


  a1.showVal( );
  a2.showVal( );


}
```

OUTPUT:

Val: 10 *a1.val*
Val: 10 *a2.val*

Val: 10 *a1.val*
Val: 20 *a2.val*

Initializing one object with another

```
int main()
{
  A  a1( 10 );
  a1.showVal( );

  A  a2(a1); // implicit copy constructor calling
  a2.showVal( );
  a2.setVal( 20 );


  a1.showVal( );
  a2.showVal( );


}
```

OUTPUT:

Val: 10 *a1.val*
Val: 10 *a2.val*

Val: 10 *a1.val*
Val: 20 *a2.val*

Copy Constructor

A copy constructor is used to initialize an object using another object of the same class

A copy constructor has the following prototype:

ClassName (const ClassName &ob);

Copy Constructor

- If we don’t define our own copy constructor, the compiler creates a default copy constructor for each class
- The default copy constructor performs member-wise copy between objects
- However, **we need to define our own copy constructor only if an object has pointers or any runtime allocation** of the resource like a file handle, a network connection, etc because the default **constructor does only shallow copy.**
- Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
- Copy constructor takes a reference to an object of the same class as an argument.

Types of Copy Constructors

It is really great way of creating new object initialisations and is still widely used by programmers.

There are two types of copy constructors which are –

- Default Copy Constructors (Does Shallow Copying)
- User Defined Copy Constructors (Does Deep Copying)

Default &User defined Copy Constructors

Default Copy Constructors –When we don’t define our own version of constructor in that case the compiler creates its own version of copy constructor. The type of copies that are made by default are called shallow copies.

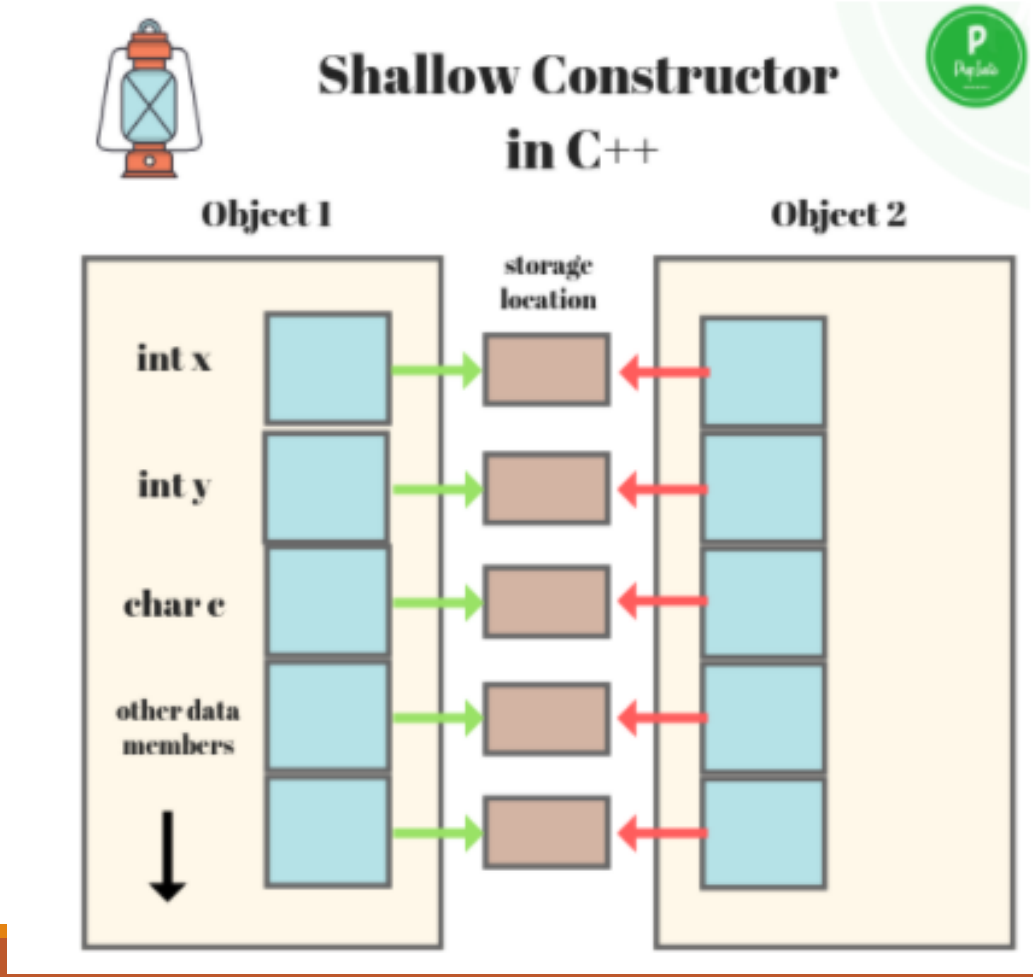
User defined copy constructor – These type of constructors are user enforced and manually created constructors.

Shallow Copy Constructor

These are in cases when a default copy constructor is created by the compiler itself. In such cases, as explained in the image, there maybe two different objects, one of which is created via default constructor called by compiler.

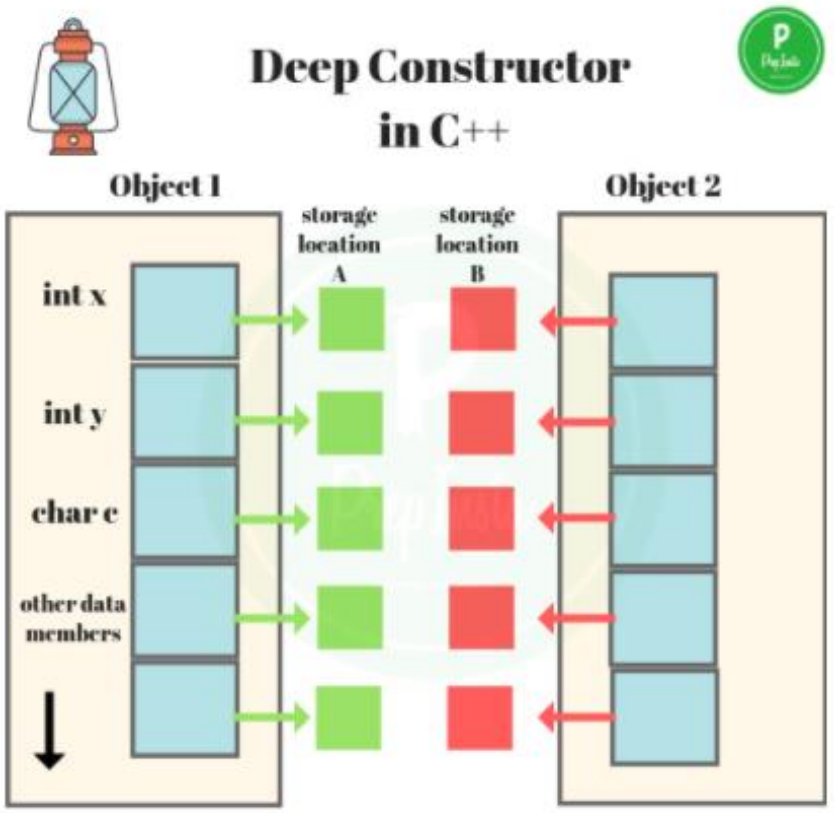
The objects do not have their individual storage locations, however, are referring to common storage location

Shallow Copy means that only the pointers will be copied not the actual resources that the pointers are pointing to. This can lead to dangling pointers if the original object is deleted.

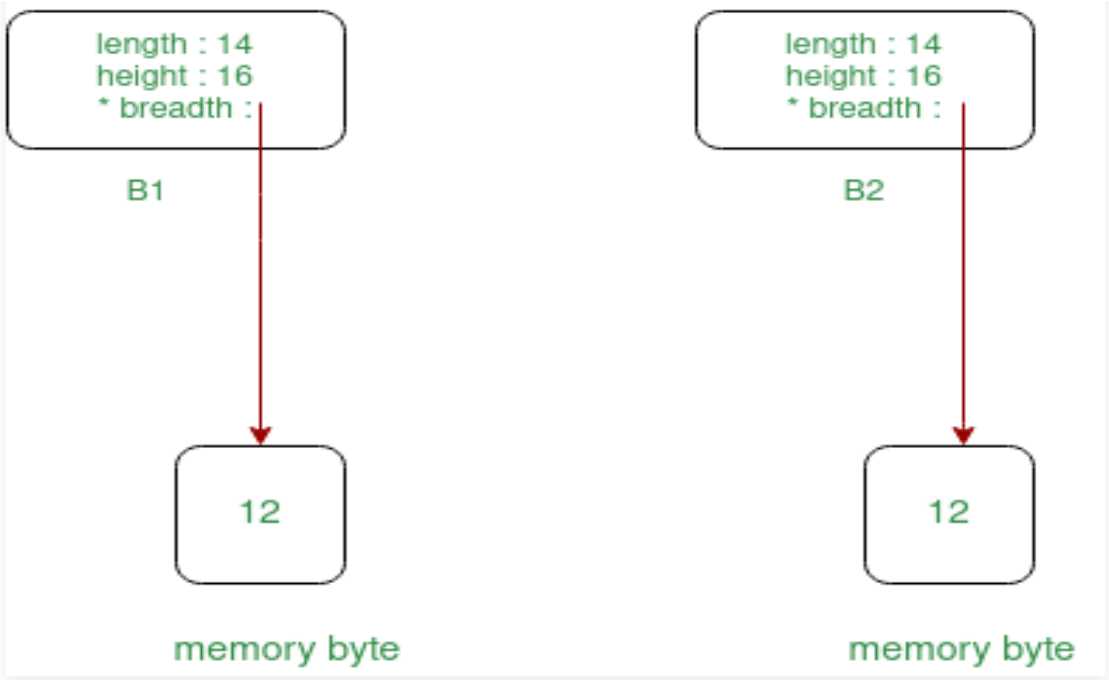


Deep Copy Constructor

- ✓Deep copy is only possible with user-defined copy constructors
- ✓In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to new copy of the dynamic resource allocated manually in the copy constructor using new operators.
- ✓At times when there is involvement of pointers and in some cases run time allocation of resources, like file handle, a network connection and others. We may encounter problem with direct assignment operator.



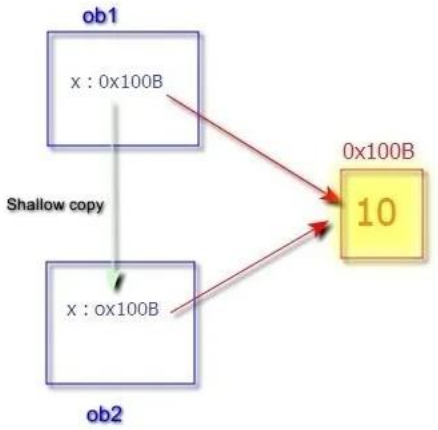
Deep Copy


length : 14
height : 16
* breadth :
B1
12
memory byte
length : 14
height : 16
* breadth :
B2
12
memory byte

Example-Shallow

```
#include <iostream>
using namespace std;
class ShalloC
{
//Sample 01: Private Data Member
private:
    int * x;
public:
    //Sample 02: Constructor with single parameter
    ShalloC(int m)
    {
        x = new int;
        *x = m;
    }
    //Sample 08: Introduce Copy Constructor and perform Deep Copy
/*  ShalloC(const ShalloC& obj)
    {
        x = new int;
        *x = obj.GetX();
    }*/
    //Sample 03: Get and Set Functions
    int GetX() const
    {
        return *x;
    }
    void SetX(int m)
    {
        *x = m;
    }
    //Sample 04: Print Function
    void PrintX()
    {
        cout << "Int X=" << *x << endl;
    }
    //Sample 05: DeAllocate the heap
    ~ShalloC()
    {
        delete x;
    }
};
int main()
{
    //Sample 06: Create Object 1 and copy that to Object 2.
    //           Print the data member for both Object 1 & 2.
    ShalloC ob1(10);
    ShalloC ob2 = ob1 ;
    ob1.PrintX();
    ob2.PrintX();
    //Sample 07: Change the Data member value of Object 1
    //           And print both Object 1 and Object 2
    ob1.SetX(12);
    ob1.PrintX();
    ob2.PrintX();
}
```

```
C:\Users\Administrator\Documents\task6.ex
Int X=10
Int X=10
Int X=12
Int X=12
```



Example-Deep

[*] task6.cpp

```
7   private:
8       int * x;
9   public:
10      //Sample 02: Constructor with single parameter
11      ShalloC(int m)
12      {
13          x = new int;
14          *x = m;
15      }
16      //Sample 08: Introduce Copy Constructor and perform Deep Copy
17      ShalloC(const ShalloC& obj)
18      {
19          x = new int;
20          *x = obj.GetX();
21      }
22      //Sample 03: Get and Set Functions
23      int GetX() const
24      {
25          return *x;
26      }
27      void SetX(int m)
28      {
29          *x = m;
30      }
31      //Sample 04: Print Function
32      void PrintX()
33      {
34          cout << "Int X=" << *x << endl;
35      }
36      //Sample 05: DeAllocate the heap
37      ~ShalloC()
38      {
39          delete x;
40      }
41  };
```

```
36      //Sample 05: DeAllocate the heap
37      ~ShalloC()
38      {
39          delete x;
40      }
41  };
42  int main()
43  {
44      //Sample 06: Create Object 1 and copy that to Object 2.
45      //           Print the data member for both Object 1 & 2.
46      ShalloC ob1(10);
47      ShalloC ob2 = ob1 ;
48      ob1.PrintX();
49      ob2.PrintX();
50      //Sample 07: Change the Data member value of Object 1
51      //           And print both Object 1 and Object 2
52      ob1.SetX(12);
53      ob1.PrintX();
54      ob2.PrintX();
55  }
```

```
C:\Users\Administrator\Documents\task6.exe
Int X=10
Int X=10
Int X=12
Int X=10

--------------------------------
Process exited after 0.04887 seconds with return value 0
Press any key to continue . . .
```

Key Differences: Pointer vs. Reference in a Copy Constructor

Feature	Pointer (`*`)	Reference (`&`)
Memory Allocation	Can be dynamically allocated (`new`)	References existing memory (no new allocation)
NULL Possibility	Can be NULL or uninitialized	Cannot be NULL, must refer to an object
Reassignment	Can change to point to another object	Cannot be reassigned after initialization
Copy Behavior	May cause a shallow or deep copy	Always binds to the original object

Feature	Shallow Copy	Deep Copy
Memory Allocation	Shares same memory	Allocates new memory
Data Independence	Modifying one affects both	Independent copies
Risk of Dangling Pointers	✅ Yes (dangerous)	❌ No (safe)
Use Case	When no dynamic memory exists	When dynamic memory exists

Discussion

Can we make copy constructor private?

Why argument to a copy constructor must be passed as a reference?

Why argument to a copy constructor should be const?

Can we make copy constructor private

Yes, you can make a copy constructor private member in C++. Making so, you are restricting someone copying your object during object creation.

objects of that class become non-copyable

A common reason to make copy constructor private is to disable default implementation of these operations.

Why argument to a copy constructor must be passed as a reference?

It is very essential to pass objects as reference. If an object is passed as value to the Copy Constructor then its copy constructor would call itself, to copy the actual parameter to the formal parameter. Thus an endless chain of call to the copy constructor will be initiated. This process would go on untill the system run out of memory.

Why argument to a copy constructor should be const?

we should use const in C++ wherever possible so that objects are not accidentally modified

Constructor overloading

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list.

task5.cpp | task6.cpp | [*] Untitled1

```cpp
// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;

class construct
{
public:
    float area;

    // Constructor with no parameters
    construct()
    {
        area = 0;
    }
    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }
        void disp()
    {
        cout<< area<< endl;
    }
};
int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

What is “**this**” keyword?

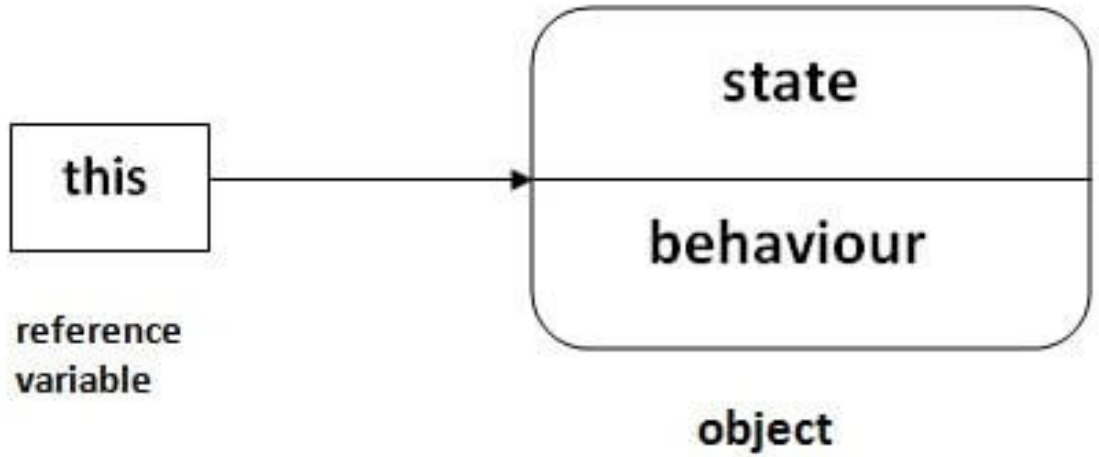
The current instance of the class is referred to by **this** keyword in C++ programming hence; this is known as this **Pointer in C++**.

This pointer holds the address of current object, in simple words you can say that this **pointer** points to the current object of the class.

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

this Pointer

this pointer is a pointer accessible only within the nonstatic member functions of a **class**,



In C++, **this** pointer is used to represent the address of an object inside a member function.

For example, consider an object obj calling one of its member function say method() as obj.method(). Then, **this** pointer will hold the address of object obj inside the member function method().

The this pointer acts as an implicit argument to all the member functions.

```
class ClassName {

    private:
      int dataMember;

    public:
       method(int a) {

  // this pointer stores the address of object obj and access dataMember

              this->dataMember = a;
            ... .. ...
        }
}

int main() {

   ClassName obj;
   obj.method(5);
   ... .. ...
 }
```

USE?

1. When local variable’s name is same as member’s name
2. To return reference to the calling object

1) When local variable's name is same as member's name

```
1  #include <iostream>
2  using namespace std;
3  class Demo {
4  private:
5    int num;
6    char ch;
7  public:
8    void setMyValues(int num, char ch){
9      this->num =num;
10     this->ch=ch;
11   }
12   void displayMyValues(){
13     cout<<num<<endl;
14     cout<<ch;
15   }
16 };
17 int main(){
18   Demo obj;
19   obj.setMyValues(100, 'A');
20   obj.displayMyValues();
21   return 0;
22 }
23
```

you can see that we have two data members num and ch. In member function setMyValues().
we have two local variables having same name as data members' name.
In such case if you want to assign the local variable value to the data members then you won't be able to do until unless you use this pointer,
because the compiler won't know that you are referring to object's data members unless you use this pointer.

1) When local variable's name is same as member's name

```
class Test{
        int x;
        public:
          Test(int x){
            x = x; }
          void display(){
            cout<<"x = "<<x;    }};
int main(){
        Test ob1(10);
        ob1.display();
        return 0; }
```

Output
x = 0

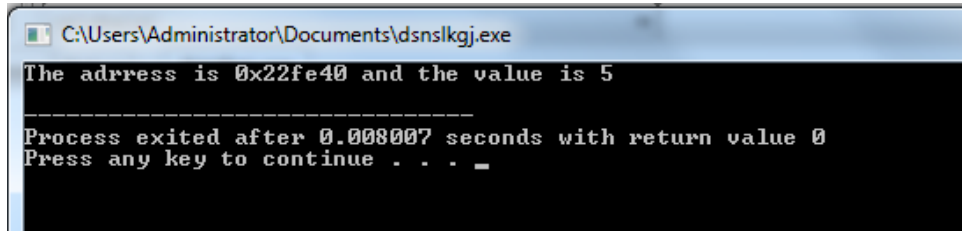
```
class Test{
        int x;
        public:
          Test(int x){
            this->x = x;
          }
          void display(){
            cout<<x; } };
int main(){
        Test ob1(10);
        ob1.display();
        return 0; }
```

Output
x = 10

1) When local variable's name is same as member's name (cont.)

dsnslkgj.cpp

```cpp
#include <iostream>
using namespace std;
#include <iostream>
using namespace std;
class example
{
private:
int x; public:
/* If function argument and data member is same then use this pointer
void set(int x)
{
    this->x = x;
}
int get()
{
return x;
}
void printAddressAndValue()
{
cout<<"The adrress is "<<this<<" and the value is "<<x<<endl;
}
};
int main(){
    example onj;
    onj.set(5);
    onj.printAddressAndValue();
}
```



2) To return reference to the calling object

To implement chain function calls, you need the reference of a calling object. You can use the "**this**" pointer to return the reference of the calling object.

Syntax

Here is the syntax:

```
Test& Test::func ()
{
   return *this;
}
```

2) To return reference to the calling object

```
#include <iostream>
using namespace std;
class Test{
    int x, y;
    public:
        Test(int x = 0, int y = 0){
            this->x = x;
            this->y = y;
        }
        Test& setX(int a){
            x = a;
            return *this;
        }
        Test& setY(int b){
            y = b;
            return *this;
        }
        void display(){
        cout<<"from "<<this<<" "<<"object have x = "<<x<<"  y = "<<y<<endl;
        }
};
int main(){
    Test ob1;
    Test ob2;
    ob1.setX(10).setY(20);
    ob1.display();//10 20        ob2.display(); // 0 0
        ob2 = ob1.setX(30);
    ob1.display();
    ob2.display();    // 30 20
    return 0;
}
```

```
from 0x6ffe10 object have x = 10  y = 20
from 0x6ffe10 object have x = 30  y = 20
from 0x6ffe00 object have x = 30  y = 20

--------------------------------
Process exited after 0.1365 seconds with return value 0
Press any key to continue . . .
```

Characteristics of the "this" Pointer

- The **"this"** pointer refers to the current instance of the object, where it allows the member function to access the object's attributes and methods.
- The **"this"** pointer is implicitly passed to all non-static member functions, where you don't need to explicitly write this in code.
- **this** points to the memory location of the current object.
- If there is a name conflict between a parameter and a member variable, this can be used to differentiate the member variable from the local parameter.
- the **"this"** pointer is constant (const), meaning it cannot be modified.
- Since **this** is a pointer, it can be dereferenced to access the current object.

What are initializer lists in C++?

The **initializer list** is used to directly initialize data members of a class. An initializer list starts after the constructor name and its parameters. The list begins with a colon (:) and is followed by the list of variables that are to be initialized – all of the variables are separated by a comma with their values in curly brackets.

Syntax: *Constructor(int var1 , int var2): Lvaraible1 (var1), Lvaraible12(var2) {}*

Syntax

```
Constructorname(datatype value1, datatype value2):datamember(value
1),datamember(value2)
{
    ...
}
```

Example

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

class Point {
public:
    int x;
    int y;
public:
    Point(int i = 0, int j = 0):x(i), y(j) {}
    /* The above use of Initializer list is optional as the
        constructor can also be written as:
        Point(int i = 0, int j = 0) {
            x = i;
            y = j;
        }
    */

    int getX() const {return x;}
    int getY() const {return y;}
};

int main() {
Point t1(10, 15);
cout<<"x = "<<t1.getX()<<", ";
cout<<"y = "<<t1.getY();
return 0;
}
```

What Are All Those Places Where Initializer List Is Must In C++?

1. For Performance reasons:
It is better to initialize all class variables in Initializer List instead of assigning values inside body.

2. For initialization of non-static const data members(we'll see this in later example)

3. For initialization of base class members : the parameterized constructor of the base class can only be called using Initializer List.(don't worry we'll use this concept in inheritance)

When do we use Initializer List in C++

4. **For initialization of reference members**

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

class Test {
    int &t;
public:
//  Test(int &t):t(t) {} //Initializer list must be used
//  int getT() { return t; }
};

int main() {
//  int x = 20;
    Test t1;
//  cout<<t1.getT()<<endl;
//  x = 30;
//  cout<<t1.getT()<<endl;
    return 0;
}
```

esources | Compile Log | Debug | Find Results | Close

	Message
\Administrator\Documents\WEEK 4.cpp	**In function 'int main()':**
Administrator\Documents\WEEK 4.cpp	[Error] use of deleted function 'Test::Test()'
Administrator\Documents\WEEK 4.cpp	[Note] 'Test::Test()' is implicitly deleted because the default definitio
Administrator\Documents\WEEK 4.cpp	[Error] uninitialized reference member in 'class Test'
Administrator\Documents\WEEK 4.cpp	[Note] 'int& Test::t' should be initialized

```cpp
using namespace std;
class Test {
    int &t;
public:
    Test(int &t):t(t) {}  //Initializer list must be used
    int getT() { return t; }
};

int main() {
    int x = 20;
    Test t1(x);
    cout<<t1.getT()<<endl;
    x = 30;
    cout<<t1.getT()<<endl;
    return 0;
}
```

Reference members must be initialized using Initializer List. In the following example, “t” is a reference member of Test class and is initialized using Initializer List

When do we use Initializer List in C++

5. **For initialization of member objects which do not have default constructor:**

```
class A {
    int i;
public:
    A(int );
};

A::A(int arg) {
    i = arg;
    cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B contains object of A
class B {
    A a;
public:
    B(int );
};

B::B(int x):a(x) { //Initializer list must be used
    cout << "B's Constructor called";
}

int main() {
    B obj(10);
    return 0;
}
```

6.When constructor’s parameter name is same as data member

```
#include <iostream>
class car{
        int a;
        public:
        car(int a):a(a) { }
        void print(){
        std::cout<<a; }
};
int main(){
        car c(10);
        c.print();
}
```

const Keyword in C++

- Constant is something that doesn't change.
- In C language and C++ we use the keyword const to make program elements constant.
- const keyword can be used in many contexts in a C++ program. It can be used with:
  - **Variables**
  - **Pointers**
  - **Function arguments**
  - **Class Data members**
  - **Class Member functions**
  - **Objects**

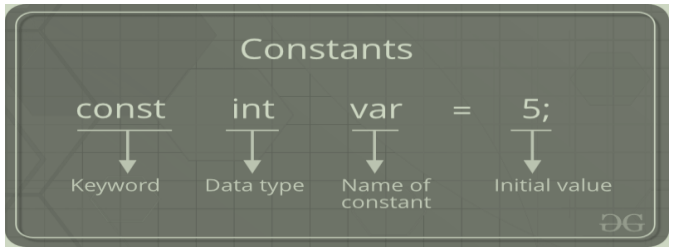
Defining Constants:

In C/C++ program we can define constants in two ways as shown below:

1. Using *#define* preprocessor directive

#define identifierName value

2. Using a *const* keyword




```
#include <iostream>
#define NUM1 10
#define NUM2  5
#define MyChar1 'E'
#define MyChar2 'N'
#define MyChar3 'D'
//#define 1My-Char
using namespace std;
int main()
{
```

Why constant?

An advantage of using constants as well as variables may be a small increase in speed when your program runs on a computer.

but it also has the advantage that it makes your programs clearer and easier to understand.

If you declare a constant you are telling the people who read your program that this value never changes within the program and so whenever it is accessed it will be the same.

The mathematical value π (pi) is a constant value and if you were to use π in your programs you wouldn't want to have to type out 3.14159 (or more decimal places) every time you needed it, nor would you ever want to give it any other value.

Constant Variables in C++

If you make any variable as constant, using const keyword, you cannot change its value. Also, the constant variables must be initialized while they are declared.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      const int b = 1;
6      b=4; // error: assignment of read-only variable 'b'
7      cout<<"here "<<b;
8     return 0;
9  }
```

```
const int arr[] = {1, 2, 3, 4, 5};
arr[0] = 10;        // will cause error
```

Function with constant Arguments

We can make the arguments of a function as const. Then we cannot change any of them.

```
void f(const int i)
{
 i++; // error
}
const int g()
{
       return 1;
}
```

Defining Class Data members as const

These are data variables in class which are defined using const keyword. They are not initialized during declaration. Their initialization is done in the constructor.

```
 7    class test
 8    {
 9        const int i; //i is a constant data member in every object
10
11        public:
12            test (int x) : i(x) {} //i can only be set through member initializer
13    };
14
15    int main()
16    {
17        test t1(1);
18
19    }
```

Member Initialization List

Constant class members can only be initialized through constructor's member initialization list

The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon

Member Initialization List

```
class A{
 const  int  x;
 const  int  y;
 public:
 A ( int val1 , int val2 )        {
    x=val1; // error
    x=val2;} // error
};
int main() {
 A  a ( 5 , 10 );}
```

Member Initialization List

```
class A{
  const  int  x;
  const  int  y;
  public:
  A ( int val1 , int val2 ) : x ( val1 ) , y ( val2 )
  {}


};
int main() {
  A  a ( 5 , 10 );}
```

Constant Parameters

```
int func(const int a, int b)
{
  a += 10;              //will cause error
  b += 20;
}
```

CONSTANT MEMBER FUNCTIONS

- Constant member function is the function that cannot modify the data members.
- To declare a constant member function, write the const keyword after the closing parenthesis of the parameter list. If there is separate declaration and definition, then the const keyword is required in both the declaration and the definition.
- Constant member functions are used, so that accidental changes to objects can be avoided. A constant member function can be applied to a non-const object.
- Keyword, const can't be used for constructors and destructors because the purpose of a constructor is to initialize data members, so it must change the object. Same goes for destructors.

Syntax:

```
return_type function_name() const;
```

CONSTANT MEMBER FUNCTIONS

```
class Demo {
    int val;
    public:
    Demo(int x = 0) {
        val = x;
    }
    int getValue() const {
     val=2; //[Error] assignment of member 'Demo::val' in read-only object
        return val;
    }
};
int main() {
    Demo d1(8);
    cout << "\nThe value using object d1 : " << d1.getValue();
    return 0;
}
```

CONSTANT OBJECTS

- As with normal variables we can also make class objects constant so that their value can't change during program execution.
- Constant objects can only call constant member functions.
- The reason is that only constant member function will make sure that it will not change value of the object.
- They are also called as read only objects.
- To declare constant object just write const keyword before object declaration.

Syntax:

```
const class_name object;
```

CONSTANT OBJECTS

```
class Demo {
public:
    int val;
    Demo(int x = 0) {
        val = x;}
    int getValue() const {
        return val; }
    int getValue1() {
        return val;}};
int main() {
    const Demo d(28);
    cout << "The value using constant object d : " << d.getValue();
    cout << "\nThe value using object d non const func : " << d.getValue1();//error
d.val=10;// error, can't modify const objects
    return 0;
}
```

CONSTANT OBJECTS

A constructor must be a non-const member function but it can still be used to initialize a const object) shows that it calls another non-const member function.

Invoking a nonconst member function from the constructor call as part of the initialization of a const object is allowed.

The “constness” of a const object is enforced from the time the constructor completes initialization of the object until that object’s destructor is called.

Constant With Pointers

There are four ways to use **const** with pointers:

- **Non-constant pointers** to **non-constant data**
- **Non-constant pointers** to **constant data**
- **Constant pointers** to **non-constant data**
- **Constant pointers** to **constant data**

Non-constant Pointers to Non-constant Data

The highest access is granted by a **non-constant pointer to non-constant data**

Data can be modified through pointer, and pointer can be made to point to other data

Non-constant Pointers With Non-constant Data

```
int main()
{
 int a = 10;
 int b = 50;

 int* pA = &a;
 *pA = 20;
 pA = &b;
}
```

Non-constant Pointers to Constant Data

Pointer can be modified to point to any other data, but the data to which it points cannot be modified through that pointer

const int * p;

Or

int const *p;

Non-constant Pointers to Constant Data

```
int main()
{
 int a = 10;
 int b = 50;

 const int* pA = &a;
 *pA = 20;                    // this line will cause error
 pA = &b;
}
```

Constant Pointers to Non-constant Data

Always points to the same memory location, but the data at that location can be modified through the pointer

```
int * const p = &val;

//here pointer is constant, so we need to assign value immediately
```

Constant Pointers to Non-constant Data

```
int main()
{
 int a = 10;
 int b = 50;

 int* const pA = &a;
 *pA = 20;
 pA = &b;                    // this line will cause error
}
```

Constant Pointers to Constant Data

Always points to the same memory location, and the data at that location cannot be modified via the pointer

const int * const p = &val;

or

int const * const p = &val;

Constant Pointers to Constant Data

```
int main()
{
 int a = 10;
 int b = 50;

 const int* const pA = &a;
 *pA = 20;                    // cannot do this
 pA = &b;                     // cannot do this as well
}
```

mutable Keyword

mutable keyword is used with member variables of class, which we want to change even if the object is of const type. Hence, mutable data members of a const objects can be modified.

Example: Mutable

```
class Zee
{
   int i;
   mutable int j;
   public:
   Zee ()
   {
      i = 0;
      j = 0;
   }
  void fool() const
   {
      i++;   // will give error
      j++;   // works, because j is mutable
   }
};

int main()
{
   const Zee obj;
   obj.fool();
}
```