

1<sup>st</sup> Semester

# ***C++ Functions***

## ***Lecture 1***

***University of Anbar***  
***College of Computer Science and Information Technology***  
***Department of Computer Science***  
***Object Oriented Programming***  
***Second Class***  
***Dr. Ruqayah R. Al-Dahhan***

# Object Oriented Programming

## Lecture 1

The conditional operator

Functions:

- default parameters
- overloading
- reference parameters

# Benefits

- **Modularity**; object maintained independently of the other objects.
- **Information-hiding**; through methods, the details remain hidden.
- **Code re-use**; re-use of object in the new program.
- **Debugging ease**; easier to spot problematic parts of program

# The conditional operator

A 'ternary' operator - it takes three arguments:

```
expr1 ? expr2 : expr3
```

If `expr1` is true then `expr2` is evaluated  
otherwise `expr3` is evaluated.

Remember logical expressions evaluate to a true or false

# The conditional operator

It allows a shorthand form of an `if` statement:

```
if (y < z) {  
    x = y;  
} else {  
    x = z;  
}
```

can also be written:

```
x = (y < z) ? y : z;
```

Sometimes this makes for more compact or efficient code.

# Functions

As a matter of style, every function ought to have its prototype declared before `main()` and its definition after `main()`.

At any rate functions **must** have at least their prototypes set up before they are used. So the compiler recognises their signatures. (signature = type, name and argument number and types)

In C++ the use of `void` is optional in a function that has no arguments:

```
int fct() { return global*2; }
```

Instead of :

```
int fct( void ) {return global*2; }
```

# Default parameters

```
int power(int n, int k = 2) {  
    //k has a default value of 2  
    if (k == 2) { return n*n; }  
    return power(n, k-1)*n;  
}
```

we can call this function in two different ways:

```
j = power(5, 3); // 5 cubed  
j = power(5);    // 5 squared
```

# Default parameters

Remarks:

Only trailing arguments may be defaulted.

More than one argument can be defaulted:

```
void fct(int j=4, int k=5, int m=7);
```

So we can call

```
fct(); or fct(1); or fct(1,2); or fct(1,2,3);
```



# Overloading

```
int max(int m, int n) { // max of two ints
    return (m>n)?m:n;
}
```

What about the max of two floats?

max is already in use -

```
float fmax(float x, float y) {
    return (x>y)?x:y;
}
```

But it adds to the clutter of our program to have a different name.

# Overloading

C++ allows functions with arguments of differing types, to have the same name.

We call this **function** overloading.

```
int max(int m, int n) {  
    return (m>n)?m:n;  
}  
float max(float x, float y) { //is OK  
    return (x>y)?x:y;  
}
```

# Overloading

An overloaded function must have parameters that differ in some way. Differing only by return type is not allowed. Otherwise the compiler would be unable to distinguish which version we mean because of the type promotion rules.

```
int triple(int a) {  
    return 3*a;  
}
```

```
float triple(int a) { //Not allowed!!  
    return 3.0*a;  
}
```

# Overloading

One function can have more arguments than another, or the types of the arguments can be different.

The compiler chooses which function to use by matching the arguments.

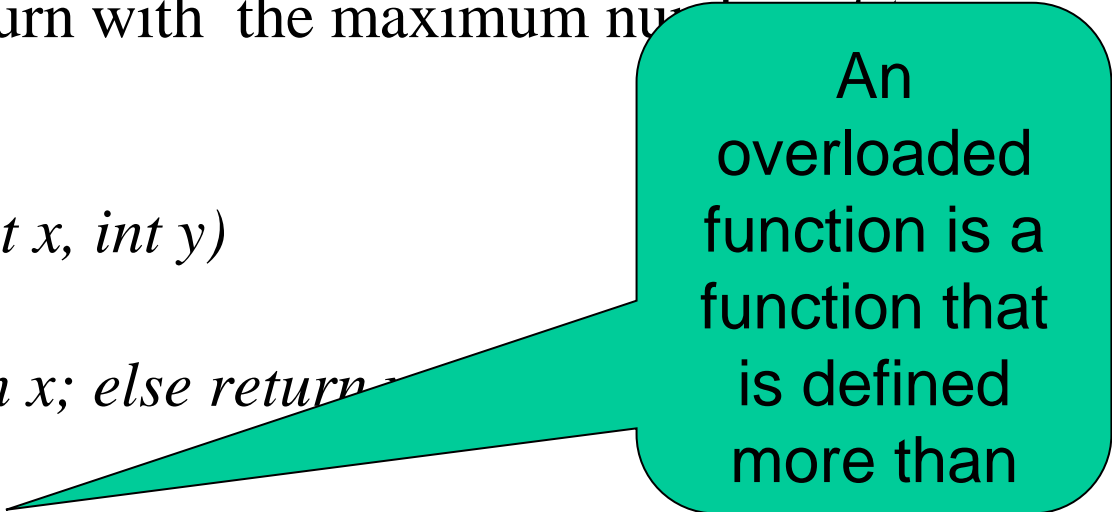
# Example #4

## Function Overloading

- Write functions to return with the maximum number of numbers

```
inline int max( int x, int y)  
{  
    if (x>y) return x; else return y;  
}
```

```
inline double max( double x, double y)  
{  
    if (x>y) return x; else return y;  
}
```



An overloaded function is a function that is defined more than once with different data types or different number of parameters

# Call-by-reference

In C++, functions can use **call-by-reference**. (A different use of the ampersand & symbol - here it says this argument is call-by-reference) In the following the printout is **j = 6**

```
int main() {  
    int j = 4;  
    changeIt(j); // j will be passed by ref  
    normal(j); // j passed as a copy  
    cout << "j = " << j << endl; // j is now 6  
}
```

```
void changeIt(int &p) {  
    p = p + 2;  
}  
void normal( int p ){  
    p =7; // only acts on passed copy  
}
```

# Call-by-reference

**p** is called '**reference parameter**'. It refers back to the original variable - so the function can alter its parameter.

```
void changeIt(int &p) {  
    p = p + 2;  
}
```

# Call-by-reference

We can create 'reference variables' similarly:

```
int main() {  
    int m;  
    int &q = m;  
    ...  
}
```

q is now just another name for the variable m.  
And we can use it to manipulate the actual variable m.



# Question

## What is the output?

```
#include<iostream>
using namespace std;
main() {
    int x=65, y=23, H;
    H=(x>y)?x+4:y-3;
    cout<<H;
}
```

# Inline Functions

- Sometimes, we use the keyword *inline* to define user-defined functions
  - Inline functions are very small functions, generally, one or two lines of code
  - Inline functions are very fast functions compared to the functions declared without the inline keyword
- Example

```
inline double degrees( double radian)
{
    return radian * 180.0 / 3.1415;
}
```

# Example #1

- Write a function to test if a number is an odd number

```
inline bool odd (int x)  
{  
    return (x % 2 == 1);  
}
```

# Example #2

- Write a function to compute the distance between two points  $(x_1, y_1)$  and  $(x_2, y_2)$

```
Inline double distance (double x1, double y1,  
                        double x2, double y2)
```

```
{  
    return sqrt(pow(x1-x2,2)+pow(y1-y2,2));  
}
```

# Example #3

- Write a function to compute  $n!$

```
int factorial( int n)  
{  
    int product=1;  
    for (int i=1; i<=n; i++) product *= i;  
    return product;  
}
```

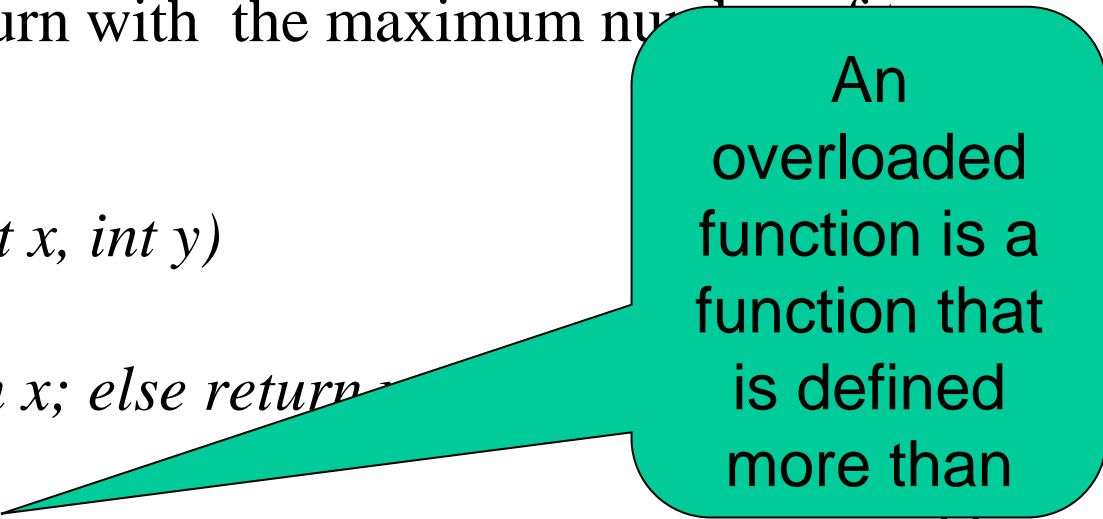
# Example #4

## Function Overloading

- Write functions to return with the maximum number of numbers

```
inline int max( int x, int y)
{
    if (x>y) return x; else return y;
}
```

```
inline double max( double x, double y)
{
    if (x>y) return x; else return y;
}
```



An overloaded function is a function that is defined more than once with different data types or different number of parameters

# Summary

Overload - to specify more than one function of the same name, but with varying numbers and types of parameters.

Reference - another name for a variable. Access to a variable via a reference is like manipulating the variable itself.

# Lecture 2

- Structure
- Classes



```

#include <iostream.h>
struct Point{
    int x;
    int y;
};
void outputAPoint( Point ); // function prototype
int main(){
    Point one, two;
    one.x = 1;
    one.y = 2;
    two.x = 3;
    two.y = 4;
    outputAPoint(one);
    outputAPoint(two);
return 0;
}

void outputAPoint( Point p ){
    cout << "Point :" << p.x << ", " << p.y << endl;
}

```

- C++ Struct syntax is simpler

- Example Output:

Point : 1,2

Point : 3,4

# Classes

A **class** is a ***user-defined type*** that contains ***data*** as well as the set of ***functions*** that manipulate that data.

# Classes

```
struct Point {  
    int x, y;  
};
```

...

```
Point w;
```

C++ implements **classes** by extending the idea of structures.

The name of a `struct` is automatically a new type.

We can use the keyword **class** instead of **struct** - they are almost the same

# Classes

In C++ a structure not only groups **data**, it also groups **operations** that can be performed on data.

```
struct Point {  
    int x,y;  
    void print() {  
        cout << "(" << x << "," << y << ")" << endl;  
    }  
};
```

We describe `print` as being a member function of the class `Point`

# Classes

`w.print()` invokes the `print` function of the `Point` structure (or class)

```
int main(){
    Point w;
    w.x = 2;
    w.y = 5;
    w.print();
}
```

# Classes

C++ limits the **visibility** of data and functions by allowing **public** and **private** parts to a structure.

By default all elements of a struct are **public**.  
Programs that use variables of this type are allowed to access all data and all functions of the structure.

```
w.y = 5;    // accessible to the calling code  
w.print();
```

Sometimes we do not want all the innards of a class to be accessible by calling code - we may want to hide part or all of it.

# Classes

Declarations within the **private** section of a structure are only visible to the structure itself.

```
struct Point {  
    public:  
        void print(void) {  
            cout << "(" << x << "," << y << ")";  
        }  
    private:  
        int x,y;  
};
```

We can no longer access the data items x and y directly from calling code!

But we are allowed to print them using `print()`!

```

struct Point {
    public:
        void print() {
            cout << "(" << x << "," << y << ")";
        }
        void init(int u, int v) {
            x = u;
            y = v;
        }
    private:
        int x,y;
};

int main() {
    Point w; // declares w to be of type Point
    w.init(2,5); // allowed, since init is public
    w.print(); // also allowed
    //w.x=90; compile ERROR since x is private
}

```



# Classes

Now the structure is very secure! - no one can alter the data of the structure without using the functions that are supplied by the structure itself:

```
int main() {
    Point w;

    w.init(2, 5);
    w.print();
    //w.x=90; ERROR x is private in Point
}
```

# Data Hiding or Encapsulation

- Why would you want to hide data from the rest of your program?
- Perhaps to protect it from accidental misuse elsewhere in the program
- Example a **Date** class might group *day*, *month*, and *year*. These need to be kept consistent - we do not want part of the user program accidentally setting *day* to something incorrect such as **-1** or even something inconsistent such as **30** when the month is **February**.
- Encapsulation lets us restrict the ways our data variables are manipulated elsewhere in the program.

# Classes

Stopping un-authorized access to data is 'good practice' and is one of the benefits of using C++.

The keywords `public` and `private` can be used many times within a structure.

It is usual to put all `public` members first and `private` members last.

Always use `private` and `public` - do not leave them as defaults.

# Classes

C++ introduces a new keyword: `class`

A `class` is exactly the same as a `struct` except that all members are private unless specified otherwise.

Most people use `class` rather than `struct`.

# Classes

```
class Point {  
    int x,y;    //private  
    void print();//private  
public:  
    void init(int, int);  
private:  
    int distance;  
};
```

```
struct Point {  
    int x,y;    //public  
    void print();//public  
public:  
    void init(int, int);  
private:  
    int distance;  
};
```

# Summary

A class is a way of implementing a data type and associated functions and operators that operate on that data.

Classes have **public** and **private** members that provide data hiding.

# Lecture 3

- Classes
- Classes that use variables of other classes
- Objects
- Static members of classes

# Classes

Functions defined within a `class` or `struct` are `inline` by default.

`inline` functions should be small, and those that are defined within a `class` should be one or two lines at most.

We can define `class` member functions outside the class definition. They are then no longer `inline` by default. Only the function prototype needs to be included within the `class`.



# Classes

```
class Point {  
    public:  
        void print(); // prototype inside class  
    private:  
        int x,y;  
};
```

```
void Point::print() { // embodiment elsewhere  
    cout << "(" << x << "," << y << ")";  
}
```

The scope resolution operator `::` is used to define functions outside the class declaration.

# Classes

```
class Point {
public:
    void print();           // not inline
    void init(int u, int v) { // inline
        x = u; y = v;
    }
private:
    int x, y;
};

void Point::print() {
    cout << "(" << x << ", " << y << ")";
}
```

# Classes

A **class** is a *user-defined type* that contains **data** as well as the set of *functions* that manipulate the data.

We often have a collection of “accessor” methods or functions - sometimes known as “get” and “set” methods or functions.

Note: Data members of a class cannot be initialized when they are declared inside the class.

These data members should be initialized using specific functions: “set” functions (like `init()` in the `Point` class).

# Classes

Member functions can also be **overloaded**.

```
class Point {
public:
    void init(int u, int v) {
        x = u; y = v;
    }
    void print();
    void print(int s);

private:
    int x, y;
};
```

# Classes

```
void Point::print() {  
    cout << "(" << x << ", " << y << " )";  
}
```

```
void Point::print(int s) {  
    cout << s;  
    print();  
}
```

# Classes

```
int main() {  
    Point w;  
    w.init(4, 7);  
    w.print();  
    cout << endl;  
    w.print(1);  
}
```

Output: (4, 7)  
(4, 7)

# Class scope

Within the second form of the print function, there is a call to the other function print (it has different arguments).

```
void Point::print(int s) {  
    cout << s;  
    print(); //No scope operator is required here.  
  
}
```

## Class scope

If there is a global function `print`, **not contained within any class**, and we want to call it within a class member function, then we use the scope operator on its own - external scope.

```
void print() {  
    cout << " The global print function";  
}
```

```
void Point::print(int s) {  
    cout << s;  
    ::print();  
}
```



# Classes can contain other classes.

```
char c;
```

```
class Y {  
    public:  
        char c;  
};
```

```
class X {  
    public:  
        char c;  
        Y y;  
};
```

```
int main () {  
    X x;  
    c = 'A';  
    x.c = 'B';  
    x.y.c = 'C';  
}
```

# Objects

C++ programming  
•is **object-oriented**- the  
programming unit is the **class**.

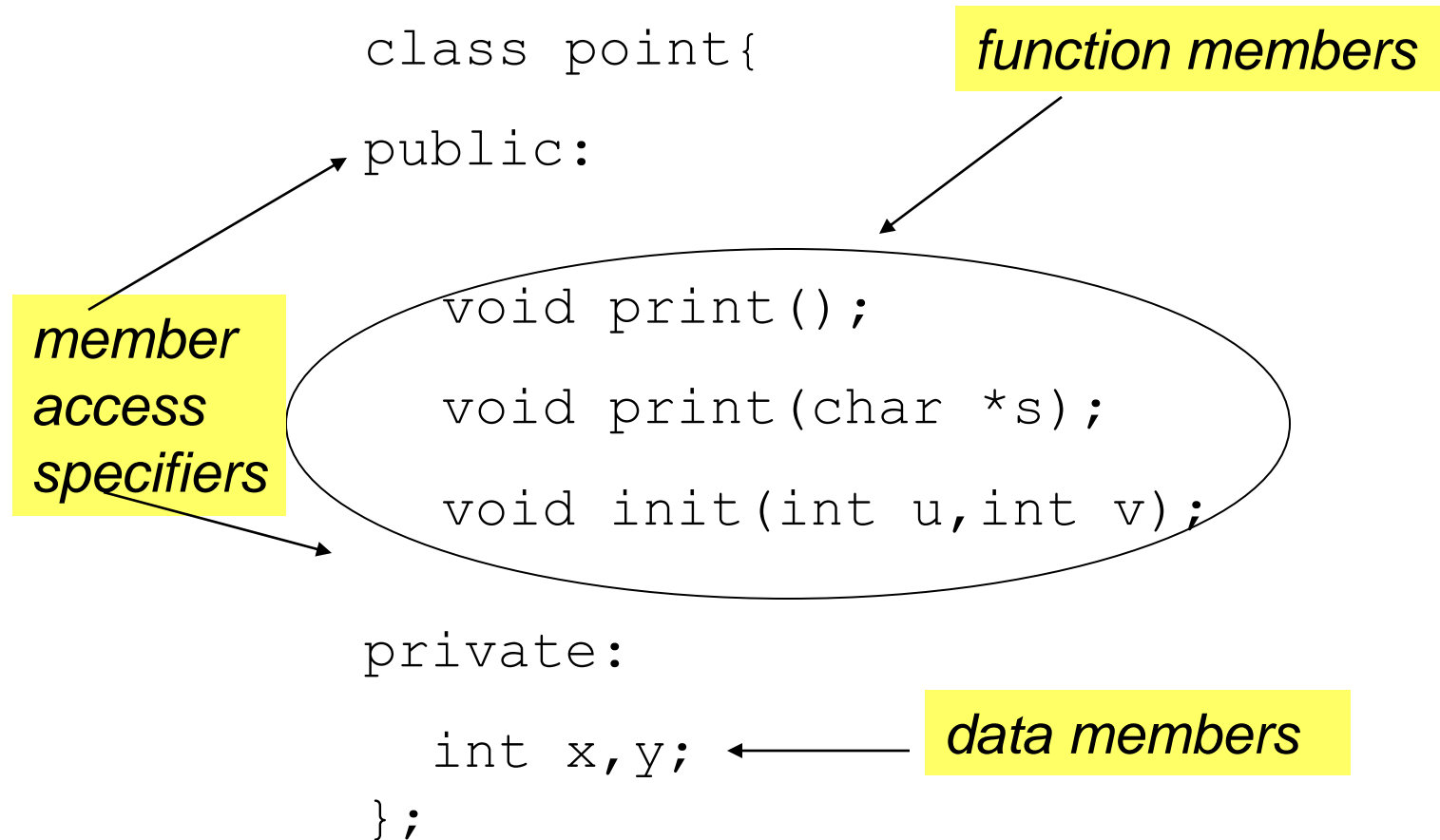
An instance of a type is called  
an **object**.

```
//j an integer object  
int j;
```

```
//w is a point object  
point w;
```

# Objects

A class is a blueprint for all its objects.



# Static members of classes

If a variable within a class is declared `static`, then there is only one instance of that variable in the program.

A `static` variable is common to **all** class variables.

(Unlike normal instance variables which are separate for each instantiation)

# Static members of classes

```
class P {
    public:
        static char c;
};

char P::c = 'W';

int main () {
    P x,y;
    cout << x.c;
    x.c = 'A';
    cout << y.c;
}
```

Correct -  
but misleading:  
`x.c` and `y.c` are the  
same thing.

# Static members of classes

It is better to refer to the static member  
as `P::c`

```
int main () {  
    P x;  
    P::c = 'A' ;  
    cout << P::c;  
}
```

# Summary

A class in C++ is a form of struct whose default access specification is private.

Classes have **public** and **private** members that provide data hiding.

The scope resolution operator `::` allows member function of various classes to have the same names as used globals.

Static data members are shared by all variables of that class type.

## Q: Define a university class as follows:

<b>Private members</b>	<b>Name, Student _No, Fees</b>
<b>Public Functions</b>	Read( ): set the private variables
	Show( ): display the variables
	Check(): check the fees if they are grater than 300\$ then write “private”, otherwise print “Suitable”
<b>Define Anbar and Baghdad as objects</b>	



# Lecture 4

Constructors

Destructors

Copy constructor

# Constructors

- Classes can have a special member function - a **constructor** - that is called when an object is created.

```
class Point {  
    public:  
    Point(int i, int j);  
    int x,y;    };  
Point::Point(int i, int j)  
    { x = i; y = j;}
```

- The constructor function has the **same name** as the class name, it has **no return type**. It is often just inline.

# Object initialization

- We now create a new point with:

Point p(4,5);

- This method has a problem though - we can't ask for an uninitialized point:

Point t;

produces an error-Point now needs two arguments.

# Object initialization

- We use **function overloading** to have several versions of the Point constructor function:

```
class Point {  
    public:  
        Point();  
        Point(int i, int j);  
    private:  
        int x,y; };  
Point::Point(){x = 0; y = 0;}  
Point::Point(int i, int j)  
    {x = i; y = j;}
```

# Object initialization

Point t; //now valid: x,y are 0,0

- A constructor with no arguments is called the **default** constructor.
- If a class does not contain **any** constructor the compiler inserts a **system default constructor** (function).

# Destructors

When an object is destroyed - the object's **destructor** is called.

If we don't free that memory before the object disappears, then the memory will never be freed - **a memory leak**. Can cause programs to crash

# Destructors

Destructors are used to release any resources allocated by the object.

Destructors are a "prepare to die" member function.

They are often abbreviated "dtor".  
Constructors are "ctor".

# Destructors - same name as class with ~ prefix

```
class Str {
    public:
        Str();
        ~Str();
    private:
        char s;
};

Str::Str()
{s = ' '; .....}

Str::~~Str() {cout<<"delete s";
.....}
```



# Destructors

A destructor:

- called by the system for you when an object is destroyable (eg about to go out of scope)
- has the same name as the class;
- with a ~ at the front;
- does not have return values;
- cannot have arguments.

# When are constructors/destructors called?

Constructors and destructors are called automatically.

The order in which they are called depends on the order in which **execution enters and leaves the scope** in which objects are instantiated and **the type of storage** for objects.

General rule: destructor calls are made in the reverse order of the constructor calls.

```
class C {
public:
    C(int);           //constructor
    ~C();            //destructor
private:
    int data;
};
C::C(int value) {
    data = value;
    cout<<"\nCtor called: "<< data;
}
C::~ ~C() {
    cout<<"\nDtor called: "<< data;
}
```

```

void createF();
C one(1); //global object
int main(){
    cout <<"Main starts here."<<endl;
    C two(2); //local object
    cout<<"After two(local)in main."<<endl;
    createF(); //f call
}

void createF(){
    cout <<endl<<" F STARTS HERE. " <<endl;
    C ten(77); //local object
    cout<< "LAST IN F. " <<endl<<endl;
}

```

Output:

Ctor called: 1

Main starts here.

Ctor called: 2

After two (local ) in main.

F STARTS HERE.

Ctor called: 77

LAST IN F.

Dtor called: 77

Dtor called: 2

Dtor called: 1

## When are constructors/destructors called?

For stack objects defined:	Constructors called:	Destructor called:
In global scope	Before any other function (including main)	When <code>main</code> terminates, or <code>exit</code> is called
Local objects	When the object enters scope.	When the object leaves scope
local objects	Once, when the object enters scope the first time.	When <code>main</code> terminates, or <code>exit</code> is called

# Problems when passing objects:

Objects can be passed as function arguments.

```
class Str{
public:
    Str() {
        p = new char[128]; assert(p!=0);
        //...
    }
    ~Str() { delete[] p }
    void print() { //... }
private:
    char* p;
};

void display( Str s);
```

# Problems when passing objects:

```
int main() {  
    Str a;  
    a.print();  
    //...  
    if (value > 100) {display(a);}  
    a.print();  
}  
void display( Str s) {s.print();}
```

Assume that the value stored in `a` was “aaaaaaaaaa”

Output:

Data is aaaaaaaaaa

Data is aaaaaaaaaa

Data is ¿?A ¿?A aa



**Problem:**

The destructor called when “display(a)” finishes damages the original object a.

How to solve it?

1. Use call by reference:     display(&a);  
so no copying is done

OR

2. Write a copy constructor for the class which  
does a proper (deep) copy

# Copy constructor

If we pass an object as an argument to a function

```
Str p;  
display(p);
```

The object is **copied** to the called routine.

The **copy constructor** of the class is called to perform the copy.

When the class does not provide its copy constructor the copy constructor provided by the system is used, **but this is not appropriate for classes with pointer data members!**

System copy constructor only does a shallow job!

# Copy constructor

The copy constructor has the form:

```
Str::Str( Str &x)
```

The original object is referred to by `x`. `Str` has to copy it into the new object.

```
Str::Str(Str &x) {  
    //.....  
}
```

This performs a **deep copy**.

# When is a copy constructor called?

1. When a copy of an object is required, such as in call-by-value.

```
Str p;  
display(p);
```

2. When returning an object by value from a function.

```
Point findMiddlePt ( Point p1, Point p2) {  
    Point temp ;  
    //---do something  
    return temp;  
}
```

3. When **initializing** an object to be a copy of another object of the same class.

```
Str x;  
//...other statements here  
Str y(x); //y declared and initiated to x
```

or

```
Str y = x; //y declared and initiated to x
```

```

#include <iostream.h>
#include <cstring.h>
#include <assert.h>
void display( Str s); // prototype

class Str{
public:
  Str(char c = ' '){
    p = new char[MAXLEN];
    assert(p!=0);
    p[0] = c;
    p[1] = '\0';
  }

  ~Str(){ delete[] p; }

  Str( const Str &src ) {           // copy constructor
    p = new char[MAXLEN]; // makes p point to separate memory
    assert( p != 0 );
    strncpy( p, src.p, MAXLEN ); // do a deep copy of the string
  }

  void print(){ cout << "Contents:" << p << endl;}

  char* p;
  const static int MAXLEN = 128;
};

void display( Str s){ // s is a partial (shallow) copy (has same internal p
  value)
  s.print();
  // s is now freed up automaticaly - but this can damage a
}

```

```

int main(){
  Str a('A'); // a now in scope
  a.print();
  int value = 101;
  cerr << "Debug line 1" << endl;
  if (value > 100){
    display(a); // passes a copy of a
  }
  cerr << "Debug line 2" << endl;
  a.print();
  cerr << "Debug line 3" << endl;
  return 0;
}

```

- Example Output:

Contents:A

Debug line 1

Contents:A

Debug line 2

Contents:A

Debug line 3

# Summary

A **constructor** constructs objects of its class type. This process may involve data members and allocating free store, using operator `new`.

A **default constructor** is a constructor requiring no arguments .

A **copy constructor** is used to copy one value into another when:

- a type variable is initialized by a type value;
- a type value is passed as an argument to a function;
- a type value is returned from a function.

A **destructor** “release any resources allocated by the object, typically by using `delete`.”

**Q:Complete the below program:**

```
#include <iostream>
```

```
using namespace std;
```

```
class Test {
```

```
    int number;
```

```
    public:
```

```
Test(int);
```

```
Test();
```

```
    int Check(); // check if the number is positive or  
negative};
```

**\*Add a destructor function to print message  
“It is Done”.**

# Lecture 5

# Friend functions / classes



## **Friend functions/classes:**

`friends` allow functions/classes access to private data of other classes.

## Friend functions

A 'friend' function has access to all 'private' members of the class for which it is a 'friend'.

To declare a 'friend' function, include its prototype within the class, preceding it with the C++ keyword 'friend'.

```
class Demo {
    friend void Change( Demo obj );
public:
    Demo(double x0=0.0, int y0=0){x=x0; y=y0;}
    void print();
private:
    double x; int y;
};
```

```
void Demo::print(){
    cout<<endl<<"This is x "<< x << endl;
    cout<<"This is y "<< y << endl;
}
```

```
void Change( Demo obj ) {
    obj.x += 100;
    obj.y += 200;
    cout<<"This is obj.x "<< obj.x << endl;
    cout<<"This is obj.y "<< obj.y << endl;
}
```

```

class T {
public:
    friend void a();
    int m();
private: // ...
};

void a() { // can access
           // private data in T...}

class S {
public:
    friend int T::m();
    //...
};

class X {
public:
    friend class T;
    //...
};

```

Global function `a()` can access  
private data in `T`

`m()` can access private data in `S`

all functions of `T` can access private  
data in `X`

friends should be used with caution:  
they by-pass C++'s data hiding  
principle.

It is the responsibility of the code for  
which access is to be given to say  
who its friends are - ie who does  
it trust!

```
# include <iostream>
Using namespace std;
class ABC;
declaration
class XYZ
{ int x ;
public :
void setvalue (int i ) { x = i; }
friend void max ( XYZ, ABC);
};
class ABC
{ int a ;
public :
void setvalue ( int i ) { a = i; }
friend void max ( XYZ, ABC);
};
```

**// Formal**

```
void max ( XYZ m, ABC n)
{
if (m.x >= n.a )
cout << m.x;
else
cout << n.a ;
}
main ( )
{ ABC abc ;
abc.setvalue ( 10 ) ;
XYZ xyz;
xyz.setvalue ( 20);
max (xyz, abc );
}
```

```

#include <iostream>
using namespace std;
const int m = 50;
class items {
int itemCode [m];
float itemPrice [m];
int count ;
public:
void CNT () { count = 0 ; }
void getitem () ;
void displaySum ( ) ;
void remove ( ) ;
void displayItems ( ) ;
};
void items :: getitem ( )
{ cout << " Enter Item code ";
cin >> itemCode [ count ] ;
cout << " Enter Item cost ";
cin >> itemPrice [ count ] ;
count++ ;
}

void items :: displaySum (void)
{ float sum = 0 ;
for (int i=0; i<count; i++)
sum = sum + itemPrice [i] ;
cout << "\n total value : " <<
sum << "\n " ;}

void items :: remove (void )
{ int a;
cout << " Enter item code : " ;
cin >> a;
for (int i=0; i< count ; i++ )
if ( itemCode [i] == a)
itemPrice [i] = 0 ;
}

void items :: displayItems (void)
{ cout << " \n code price \n " ;
for (int i=0; i<count; i++)
{
main ( )
{ items order ;
order.CNT ( ) ;
int x ;
do
{ cout <<" you can do the
following: "
<< " Enter appropriate
number \n " ;
cout << " \n 1: Add an item " ;
cout << " \n 2: Display total
value" ;
cout << " \n 3: Delete an item " ;
cout << " \n 5: Quit " ;
cout << " \n \n What is your
option ? " ;
cin >> x ;
switch (x)
{ case 1: order.getitem ( ) ; break;
case 2: order.displaySum ( ) ;

```

```

#include <iostream>
using namespace std;
const int m = 50;
class items {
int itemCode [m];
float itemPrice [m];
int count ;
public:
void CNT () { count = 0 ; }
void getitem () ;
void displaySum ( ) ;
void remove ( ) ;
void displayItems ( ) ;
};
void items :: getitem ( )
{ cout << " Enter Item code ";
cin >> itemCode [ count ] ;
cout << " Enter Item cost ";
cin >> itemPrice [ count ] ;
count++ ;
}

void items :: displaySum (void)
{ float sum = 0 ;
for (int i=0; i<count; i++)
sum = sum + itemPrice [i] ;
cout << "\n total value : " <<
sum << "\n " ;}

void items :: remove (void )
{ int a;
cout << " Enter item code : " ;
cin >> a;
for (int i=0; i< count ; i++ )
if ( itemCode [i] == a)
itemPrice [i] =
0 ;
}

void items :: displayItems (void)
{ cout << " \n code price \n " ;
for (int i=0; i<count; i++)
{
main ( )
{ items order ;
order.CNT ( ) ;
int x ;
do
{ cout <<" you can do the
following: "
<< " Enter appropriate
number \n " ;
cout << " \n 1: Add an item " ;
cout << " \n 2: Display total
value" ;
cout << " \n 3: Delete an item " ;
cout << " \n 5: Quit " ;
cout << " \n \n What is your
option ? " ;
cin >> x ;
switch (x)
{ case 1: order.getitem ( ) ; break;
case 2: order.displaySum ( ) ;

```

```

#include <iostream>
using namespace std;
const int m = 50;
class items {
int itemCode [m];
float itemPrice [m];
int count ;
public:
void CNT () { count = 0 ; }
void getitem () ;
void displaySum ( ) ;
void remove ( ) ;
void displayItems ( ) ;
};
void items :: getitem ( )
{ cout << " Enter Item code ";
cin >> itemCode [ count ] ;
cout << " Enter Item cost ";
cin >> itemPrice [ count ] ;
count++ ;
}

void items :: displaySum (void)
{ float sum = 0 ;
for (int i=0; i<count; i++)
sum = sum + itemPrice [i] ;
cout << "\n total value : " <<
sum << "\n " ;}

void items :: remove (void )
{ int a;
cout << " Enter item code : " ;
cin >> a;
for (int i=0; i< count ; i++ )
if ( itemCode [i] == a)
itemPrice [i] = 0 ;
}

void items :: displayItems (void)
{ cout << " \n code price \n " ;
for (int i=0; i<count; i++)
{
switch (x)
{ case 1: order.getitem () ; break;
case 2: order.displaySum () ;
}
}
}

main ( )
{ items order ;
order.CNT ( ) ;
int x ;
do
{ cout <<" you can do the
following: "
<< " Enter appropriate
number \n " ;
cout << " \n 1: Add an item " ;
cout << " \n 2: Display total
value" ;
cout << " \n 3: Delete an item " ;
cout << " \n 5: Quit " ;
cout << " \n \n What is your
option ? " ;
cin >> x ;
}
}

```



```

#include <iostream>
using namespace std;
class employee
{
    char name [ 30 ] ;
    float age ;
public :
    void getData (void );
    void putData (void);
};
void employee :: getData (void)
{ cout << " Enter name : " ;
  cin >> name ;
  cout << " Enter age: " ;
  cin >> age ; }

```

```

void employee :: putData (void )
{
  cout << " name : " << name << " \n " ;
  cout << " Age : " << age << " \n" ;
}
const int size = 3 ;
main ( )
{ employee manager [ size ] ;
  for (int i= 0; i < size ; i ++ )
  {
    manager [ i ].getData ( ) ;
  }
  cout << " \n " ;
  for ( int i = 0; i < size; i ++ )
  {
    cout << " \n manager " << i+1 << " \n
";
    manager [ i ].putData ( ) ;  } }

```

# Example 1

```
#include <iostream>
using namespace std;
class integer
{
    int m , n ;

public :
    integer (int , int) ; // constructor

declared
    void display()
    {
        cout << "m = " << m << "\n";
        cout << "n = " << n << "\n";
    }

};
integer :: integer (int x , int y)
{ m = x ; n = y ;}
```

```
void main( )
{
    integer int1 (0 , 100 ); // implicit call
    integer int2 = integer(25 , 75); //explicit call
    cout << "\n OBJECT1 " << "\n";
    int1.display( );
    cout << "\n OBJECT2" << "\n";
    int2.display( );
}
```

# Example2

```
#include <iostream.h>
class complex
{
    float x , y;
public:
    complex( ) { }          //satisfy
    compiler
    complex (float a) {x = y = a ;}
    complex (float real , float imag)
    {x = real ; y = imag; }
    friend complex sum (complex ,
complex);
    friend void show (complex);
};
```

```
complex sum (complex c1 , complex c2)
{
    complex c3 ;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return c3;
}
```

```
void show (complex c)
{cout << c.x << "+j " << c.y << "\n"; }
```

```
main( )
{
    complex A (2.7 , 3.5);
    complex B (1.6);
    complex C;
    C = sum (A , B);
    cout << "A = " ; show (A);
    cout << "B = " ; show (B);
    cout << "C = " ; show (C);
}
```

# Copy Constructor

```
#include<iostream>
using namespace std;
class code{
    int id;
public:
    code(){ }
    code(int a)
    {id = a;}
    code (code &x) //x alias for
object A .
    {          id = x.id; }

    void display()
    {          cout<<id; }
};
```

```
main( )
{
    code A(100); // Object A is
created and initialized
    code B(A); // Copy
Constructor Called
    code C = A;
    code D;
    D=B;
    cout<<"\n id of A: ";A.display( );
    cout<<"\n id of B: ";B.display( );
    cout<<"\n id of C: ";C.display( );
    cout<<"\n id of D: ";D.display( );
}
```

# Summary

A constructor constructs objects of its class type. This process may involve data members and allocating free store, using operator `new`.

A default constructor is a constructor requiring no arguments  
.

A copy constructor is used to copy one value into another when:

- a type variable is initialized by a type value;
- a type value is passed as an argument to a function;
- a type value is returned from a function.

A destructor “release any resources allocated by the object, typically by using `delete`.”