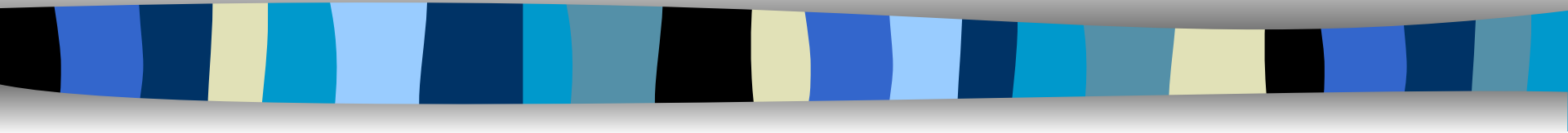


Objects and Classes





Object Oriented Programming

- Object-oriented programming (OOP)
 - Encapsulates data (attributes) and functions (behavior) into packages called classes.
- So, Classes are user-defined (programmer-defined) types.
 - Data (data members)
 - Functions (member functions or methods)
- In other words, they are structures + functions



Inline Functions

inline function header

```
{ function body  
}
```

// A program illustrating inline function #include<iostream.h> #include<conio.h>

```
inline int max(int x, int y)
```

```
{  
    if(x>y)  
        return x;  
    else return y;  
}
```

```
int main( ) {
```

```
    int a,b;  
    cout<<"enter two numbers";  
    cin>>a>>b;  
    cout << "The max is: " <<max(a,b) << endl;  
    return 0;  
}
```



Macros Vs inline functions

- Inline functions follow all the protocols of type safety enforced on normal functions.
- Inline functions are specified using the same syntax as any other function except that they include the inline keyword in the function declaration.
- Expressions passed as arguments to inline functions are evaluated once.
- In some cases, expressions passed as arguments to macros can be evaluated more than once.

macros are expanded at pre-compile time, you cannot use them for debugging, but you can use inline functions



Inline Functions

Limitations of macros

Can increase fun size so much that it may not fit in cache

Where of inline functions may not work:

1. For fun returning values, if loop,a switch,or goto exist.
2. For fun not returning values, if return stmt exists
3. If fun contain static var
4. If inline fun are recursive

The benefits of inline functions are as follows :

- 1. Better than a macro.
- 2. Function call overheads are eliminated.
- 3. Program becomes more readable.
- 4. Program executes more efficiently.



Default arguments

C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call.

cannot provide a default value to a particular argument in the middle of an argument list. When you create a function that has one or more default arguments, those arguments must be specified only once: either in the function's prototype or in the function's definition if the definition precedes the function's first use.

Ex. Bank interest

Example

```
#include <iostream.h> #include<conio.h>
int sum(int a, int b=20){ return( a + b);
}
int main (){
int a = 100, b=200, result;
result = sum(a, b);    //here a=100 , b=200 cout << "Total value is :" << result << endl;
result = sum(a); //here a=100 , b=20(using default value) cout << "Total value is :" << result
<< endl;
return 0;
}
```



Constant Arguments

- In C++ ,These arguments(s) is/are treated as constant(s). These values cannot be modified by the function.
- For making the arguments(s) constant to a function, we should use the keyword **const** as given below in the function prototype :

Void max(const float x, const float y, const float z);

- Here, the qualifier **const** informs the compiler that the arguments(s) having **const** should not be modified by the function max (). These are quite useful when call by reference method is used for passing arguments.



Function Overloading

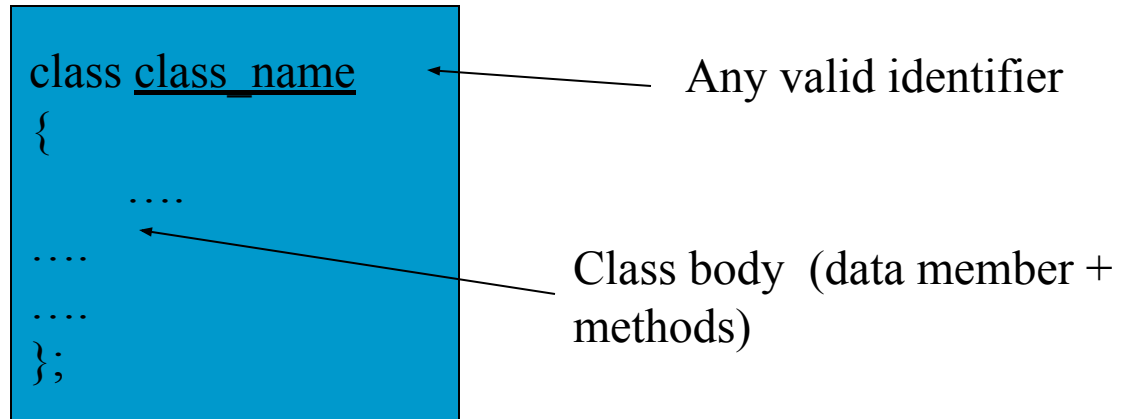
- It is the process of using the same name for two or more functions.
- Each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation.
- Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name.
- To overload a function, simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and/or type of the arguments used to call the function.
- Two functions differing only in their return types cannot be overloaded.
- **Example**

Example

- `#include<iostream.h>`
- `#include<conio.h>`
- `int sum(int p,int q,int r);`
- `double sum(int l,double m);`
- `float sum(float p,float q);`
- `int main()`
- `{`
- `cout<<"sum="<< sum(11,22,33); //calls func1`
- `cout<<"sum="<< sum(10,15.5); //calls func2`
- `cout<<"sum="<< sum(13.5,12.5); //calls func3 return 0;`
- `}`
- `int sum(int p,int q,int r)`
- `{//func1 return(p+q+r);`
- `}`
- `double sum(int l,double m)`
- `{//func2 return(l+m);`
- `}`
- `float sum(float p,float q)`
- `{//func3 return(p+q);`
- `}`

Classes in C++

- A class definition begins with the keyword *class*.
- The body of the class is contained within a set of braces, `{ }`; (notice the semi-colon).





Class

- The class is the **cornerstone** of C++
 - It makes possible encapsulation, data hiding and inheritance

Type

- Concrete representation of a concept
 - Eg. **float** with operations like -, *, + (math real numbers)

Class

- A user defined type
- Consists of both data and methods
- Defines properties and behavior of that type

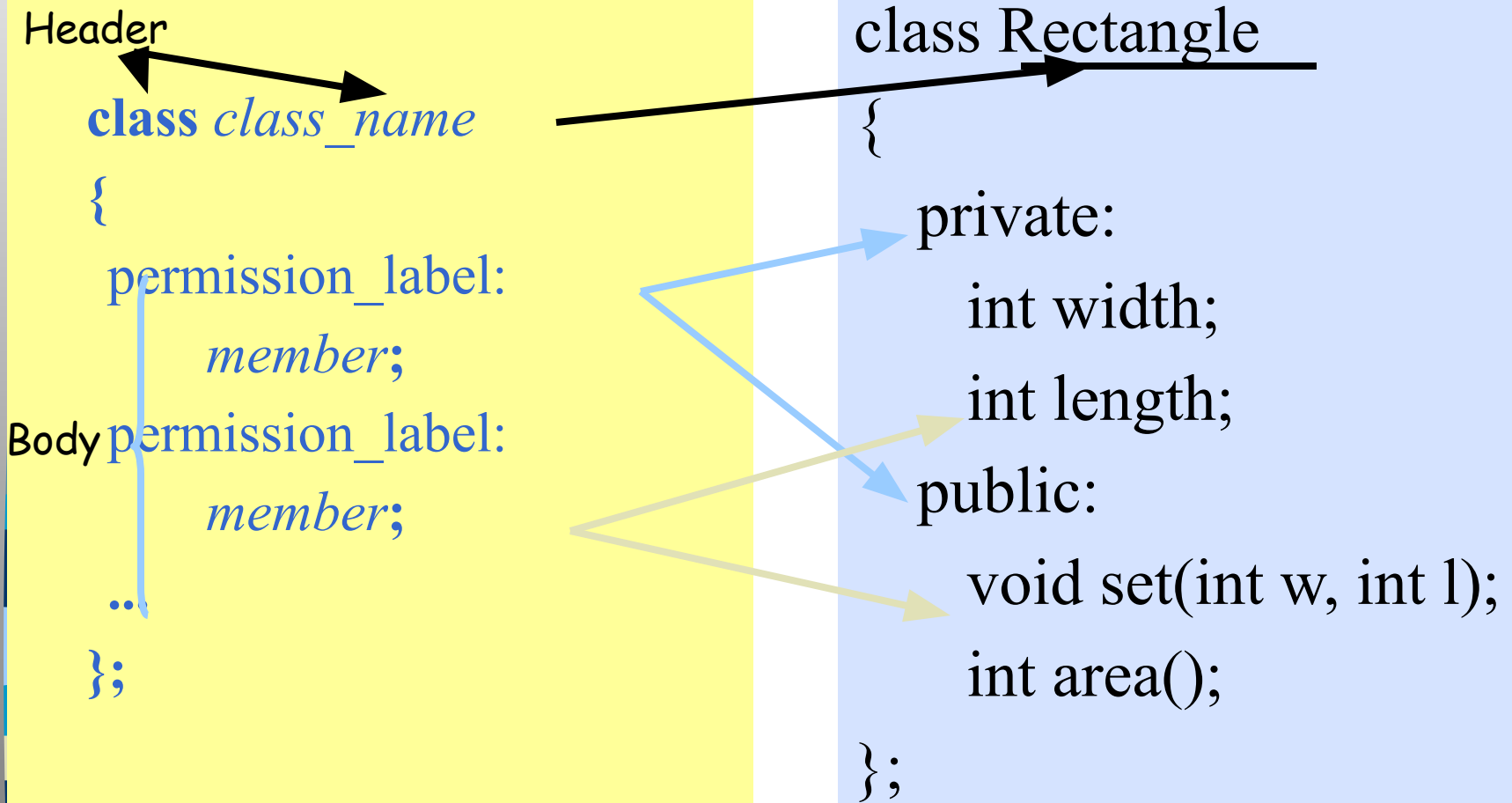
Advantages

- Types matching program concepts
 - Game Program (Explosion type)
- Concise program
- Code analysis easy
- Compiler can detect illegal uses of types

Data Abstraction

- Separate the implementation details from its essential properties

Define a Class Type



Classes in C++

```
class class_name
```

```
{
```

```
    private:
```

```
        ...
```

```
        ...
```

```
        ...
```

```
    public:
```

```
        ...
```

```
        ...
```

```
        ...
```

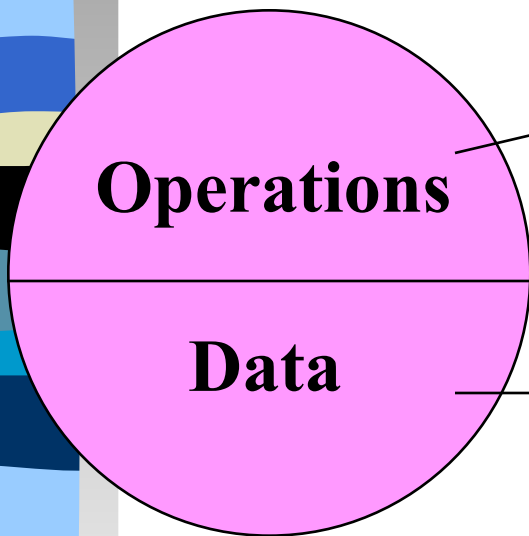
```
};
```

private members or methods

Public members or methods

What is an object?

OBJECT



Operations

**set of methods
(member functions)**

Data

**internal state
(values of private data members)**

Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

r1 is statically allocated

```
main()
{
    Rectangle r1;
    → r1.set(5, 8);
}
```

r1

**width = 5
length = 8**

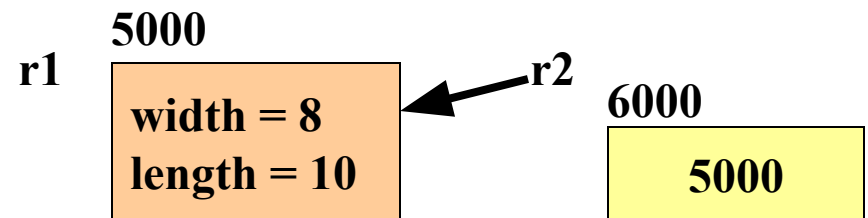
Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

r2 is a pointer to a Rectangle object

```
main()
{
    Rectangle r1;
    r1.set(5, 8);           //dot notation

    Rectangle *r2;
    r2 = &r1;
    r2->set(8,10);          //arrow notation
}
```



Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

r3 is dynamically allocated

```
main()
{
    Rectangle *r3;
    r3 = new Rectangle();

    r3->set(80,100);    //arrow notation

    delete r3;
    → r3 = NULL;
}
```

r3
6000
NULL

Object Initialization

1. By Assignment

```
#include <iostream.h>

class circle
{
public:
    double radius;
};
```

```
int main()
{
    circle c1;           // Declare an instance of the class circle
    c1.radius = 5;       // Initialize by assignment
}
```

- Only work for public data members
- No control over the operations on data members

Object Initialization

```
#include <iostream.h>
```

```
class circle
```

```
{
```

```
private:
```

```
    double radius;
```

```
public:
```

```
    void set (double r)
```

```
    {radius = r;}
```

```
    double get_r ()
```

```
    {return radius;}
```

```
};
```

2. By Public Member Functions

```
int main(void) {
```

```
    circle c;           // an object of circle class
```

```
    c.set(5.0);         // initialize an object with a public member function
```

```
    cout << "The radius of circle c is " << c.get_r() << endl;
```

```
    // access a private data member with an accessor
```

```
}
```

Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
}
```

r2 is a pointer to a Rectangle object

```
main()
{
    Rectangle r1;
    r1.set(5, 8);           //dot notation

    Rectangle *r2;
    r2 = &r1;
    r2->set(8,10);          //arrow notation
}
```

r1 and r2 are both initialized by public member function set

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        Rectangle();
        Rectangle(const Rectangle &r);
        Rectangle(int w, int l);
        void set(int w, int l);
        int area();
}
```

3. By Constructor

- Default constructor
- Copy constructor
- Constructor with parameters

They are publicly accessible

Have the same name as the class

There is no return type

Are used to initialize class data members

They have different signatures

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

When a class is declared with no constructors, the compiler automatically assumes **default** constructor and **copy** constructor for it.

- Default constructor

```
Rectangle :: Rectangle() { };
```

- Copy constructor

```
Rectangle :: Rectangle (const Rectangle &
    r)
{
    width = r.width; length = r.length;
};
```

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
}
```

- Initialize with **default** constructor

```
Rectangle r1;
Rectangle *r3 = new Rectangle();
```

- Initialize with **copy** constructor

```
Rectangle r4;
r4.set(60,80);

Rectangle r5 = r4;
Rectangle r6(r4);

Rectangle *r7 = new Rectangle(r4);
```

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        Rectangle(int w, int l)
        {width =w; length=l;}
        void set(int w, int l);
        int area();
}
```

If any constructor with any number of parameters is declared, no **default** constructor will exist, unless you define it.

```
Rectangle r4;// error
```

- Initialize with constructor

```
Rectangle r5(60,80);
```

```
Rectangle *r6 = new Rectangle(60,80);
```

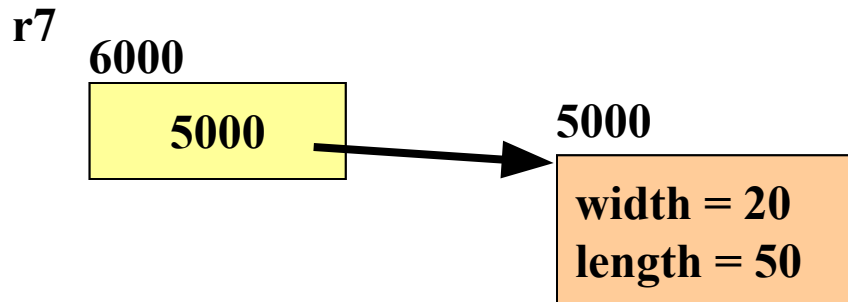

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        Rectangle();
        Rectangle(int w, int l);
        void set(int w, int l);
        int area();
}
```

Write your own constructors

```
Rectangle :: Rectangle()
{
    width = 20;
    length = 50;
};
```

```
Rectangle *r7 = new Rectangle();
```



Object Initialization

```
class Account
{
    private:
        char *name;
        double balance;
        unsigned int id;
    public:
        Account();
        Account(const Account &a);
        Account(const char *person);
}
```

```
Account :: Account()
{
    name = NULL; balance = 0.0;
    id = 0;
};
```

With constructors, we have more control over the data members

```
Account :: Account(const Account &a)
{
    name = new char[strlen(a.name)+1];
    strcpy (name, a.name);
    balance = a.balance;
    id = a.id;
};
```

```
Account :: Account(const char *person)
{
    name = new char[strlen(person)+1];
    strcpy (name, person);
    balance = 0.0;
    id = 0;
};
```

Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

```
main()
{
    Rectangle r1;
    Rectangle r2;

    r1.set(5, 8);
    cout<<r1.area()<<endl;

    r2.set(8,10);
    cout<<r2.area()<<endl;
}
```

Demonstrates a small, simple object

```
// smallobj.cpp
```

```
#include <iostream>
```

```
class smallobj
```

```
{
```

```
private:
```

```
    int somedata;      → Data members
```

```
public:
```

```
    void setdata(int d)
```

```
    {
```

```
        somedata = d;
```

```
    }
```

→ Member Functions

```
    void showdata()
```

```
    {
```

```
        cout << "Data is : " << somedata << endl;
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    smallobj s1, s2;
```

```
    s1.setdata(1106);
```

```
    s2.setdata(1425);
```

```
    s1.showdata();
```

```
    s2.showdata();
```

```
    return 0;
```

```
}
```

→ Calling Member Functions

Output of the Program:

Data is : 1106

Data is : 1425



Classes and Objects

- An object has the same relationship to a class that a variable has to a data type
- An object is said to be an instance of a class
 - Chevrolet is instance of a vehicle
 - In smallobj example, s1, s2 are instances of smallobj class



Functions are public, Data is private

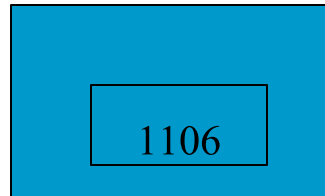
- Usually the data within a class is private and functions are public
- Data is hidden so it will be safe from accidental manipulation
- Functions that operate on the data are public so they can be accessed from outside the class
- However, there is no rule that says data must be private and functions must be public



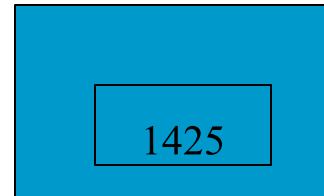
Objects

Objects of class `smallobj`

s1



s2



```
smallobj s1, s2;  
s1.setdata(1106);  
s2.setdata(1425);
```



```
// Objpart.cpp
```

```
#include <iostream>
```

```
#include <conio.h>
```

```
class part
```

```
{
```

```
private:
```

```
    int modelnumber;
```

```
    int partnumber;
```

```
    float cost;
```

```
public:
```

```
    void setpart(int mn, int pn, float c)
```

```
    {
```

```
        modelnumber = mn;
```

```
        partnumber = pn;
```

```
        cost = c;
```

```
    }
```

```
    void showpart()
```

```
    {
```

```
        cout << "Model: " << modelnumber << endl;
```

```
        cout << "Part: " << partnumber << endl;
```

```
        cout << "Cost: " << cost << endl;
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    part part1;
```

```
    part1.setpart(6244,329,55);
```

```
    part1.showpart();
```

```
    getch();
```

```
    return 0;
```

```
}
```




Classes in C++

- Within the body, the keywords *private:* and *public:* specify the access level of the members of the class.
 - the default is *private*.
- Usually, the data members of a class are declared in the *private:* section of the class and the member functions are in *public:* section.
- Don't confuse data hiding with the security techniques used to protect computer databases



Class Definition - Access Control

- **Information hiding**
 - To prevent the internal representation from direct access from outside the class
- **Access Specifiers**
 - **public**
 - may be accessible from anywhere within a program
 - **private**
 - may be accessed only by the member functions, and friends of this class
 - **protected**
 - acts as public for derived classes
 - behaves as private for the rest of the program



Class Definition

Data Members

- Can be of any type, built-in or user-defined

- non-static data member*

- Each class object has its own copy

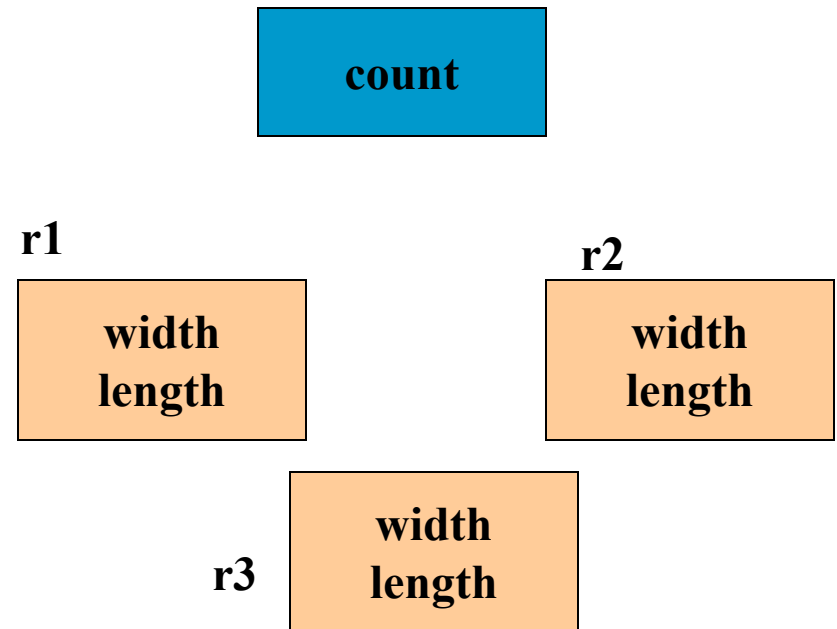
- static data member*

- Acts as a global variable
 - One copy per class type, e.g. counter

Static Data Member

```
class Rectangle
{
    private:
        int width;
        int length;
    → static int count;
    public:
        void set(int w, int l);
        int area();
}
```

```
Rectangle r1;
Rectangle r2;
Rectangle r3;
```





Class Definition

Member Functions

- Used to

- access the values of the data members (**accessor**)
- perform operations on the data members (**implementor**)

- Are declared inside the class body

- Their definition can be placed inside the class body, or outside the class body

- Can access both public and private members of the class

- Can be referred to using dot or arrow member access operator

Define a Member Function

1. Outside class definition

2. Inside class definition

return_type functionName(parameter_list)

{

function body;

}

class Rectangle

{

private:

int width, length;

public:

void set (int w, int l);

~~int area() {return width*length; }~~

};

class name

member function name

inline

r1.set(5,8);

rp->set(8,10);

void Rectangle :: set (int w, int l)

{

width = w;

length = l;

}

scope operator

2. Inside class definition

```
class book {  
    char title[30];  
    float price;  
    public:  
        void getdata(char [],float); // declaration  
        void putdata()//definition inside the class {  
            cout<<"\nTitle of Book: "<<title;  
            cout<<"\nPrice of Book: "<<price;  
        } ;  
};
```

the member function putdata() is defined inside the class book. Hence, putdata() is by default an inline function.

Note :functions defined outside the class can be explicitly made inline by prefixing the keyword inline before the return type of the function in the function header. For example, consider the definition of the function getdata().

```
1 inline void book ::getdata (char a [],float b)  
2 {  
3     body of the function  
}
```

Nesting of member function

A member function can be called by using its name inside another member function of the same class called Nesting of member function.

//program:adding of two object

```
#include<iostream>
```

```
using namespace std;
```

```
class add
```

```
{
```

```
    public:
```

```
        void get();
```

```
        void display();
```

```
};
```

```
void add ::get()
```

```
{
```

```
    cout<<"Enter x and y;
```

```
    cin>>x>>y;
```

```
    display();
```

```
}
```

```
void add ::display()
```

```
{
```

```
    cout<<"z="<<z;
```

```
}
```

```
int main()
```

```
{
```

```
    add ob;
```

```
    ob.get();
```

```
    ob.display();
```

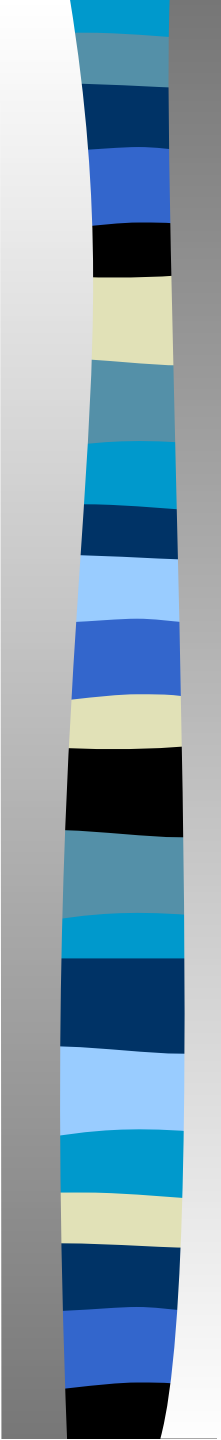
```
    return 0;
```

```
}
```

output:

Enter x and y

z=32765 z=32765



You should make a function private when you don't need other objects or classes to access the function, when you'll be invoking it from within the class.

const member function

- **const member function**

- **declaration**

- *return_type func_name (para_list) const;*

- **definition**

- *return_type func_name (para_list) const { ... }*
- *return_type class_name :: func_name (para_list) const { ... }*

- **Makes no modification about the data members (safe function)**

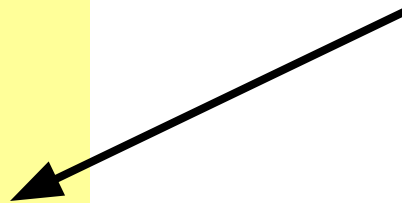
- **It is illegal for a const member function to modify a class data member**

Const Member Function

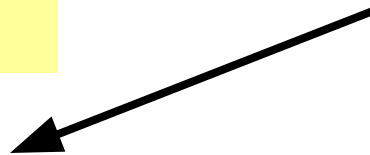
```
class Time
{
    private :
        int    hrs, mins, secs ;

    public :
        void    Write ( ) const ;
} ;
```

function declaration



function definition



```
void Time :: Write ( ) const
{
    cout << hrs << ":" << mins << ":" << secs << endl;
}
```



C++ objects as Data Types

```
class Distance
```

```
{
```

```
private:
```

```
    int feet;
```

```
    int inches;
```

```
public:
```

```
    void setdist(int f, int i)
```

```
    {
```

```
        feet = f;
```

```
        inches = i;
```

```
    }
```

```
    void showdist()
```

```
    {
```

```
        cout << feet << "\"" << inches << "\"" << endl;
```

```
    }
```

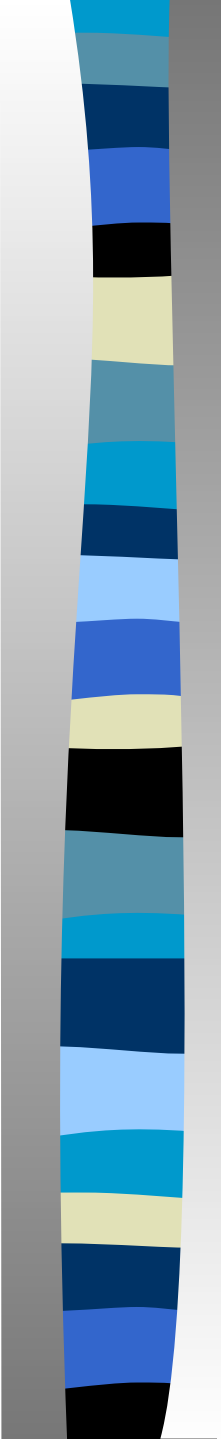
```
};
```



Object as Function Arguments

```
class Distance
{
private:
    int feet;
    float inches;

public:
    void getdist()
    {
        cout << "Enter Feet: " ;
        cin >> feet;
        cout<< "Enter Inches: ";
        cin>> inches;
    }
}
```



```
void setdist(int f, float i)
```

```
{
```

```
    feet = f;
```

```
    inches = i;
```

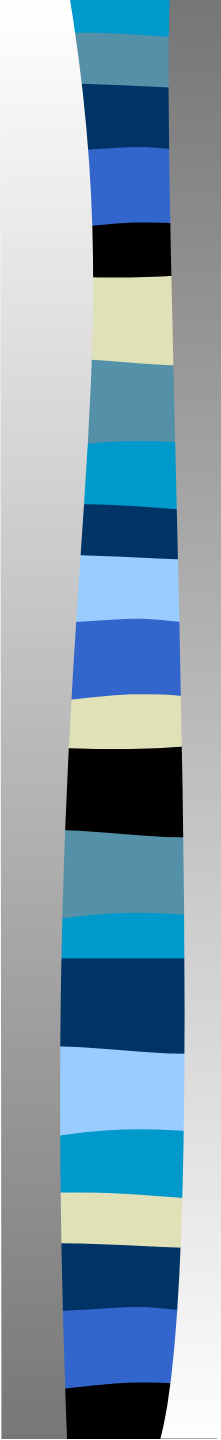
```
}
```

```
void showdist()
```

```
{
```

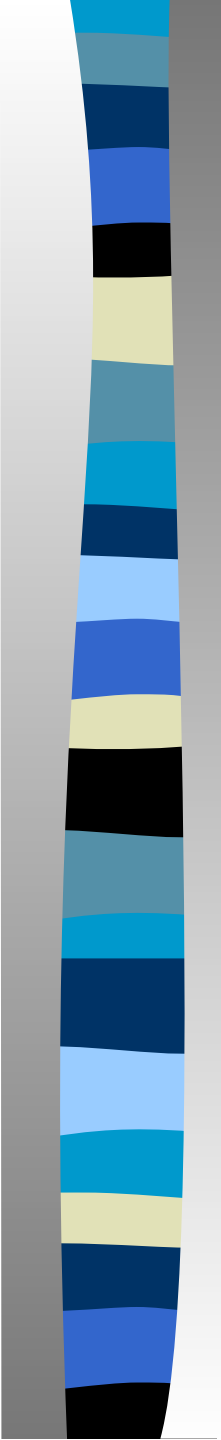
```
    cout << feet << "\" " << inches << "\" " << endl;
```

```
}
```



```
void add_dist(Distance d2, Distance d3)
{
    inches = d2.inches + d3.inches;
    feet = 0;
    if(inches >=12.0)
    {
        inches -= 12.0;
        feet++;
    }
    feet += d2.feet + d3.feet;
}

; // class ends
```



```
int main()
{
    Distance dist1, dist2, dist3;
    dist2.setdist(11,6.25);
    dist1.getdist();
    dist3.add_dist(dist1,dist2);
    dist1.showdist();
    dist2.showdist();
    dist3.showdist();
    return 0;
}
```




Let's Revise Constructor

C++ Classes

C++ classes may be instantiated either *automatically* (on the stack):

```
MyClass oVal;           // constructor called  
                          /  
// destroyed when scope ends
```

dynamically (in the heap)

```
MyClass *oPtr;           // uninitialized  
pointer  
  
oPtr = new MyClass;      // constructor called  
                          //  
// must be explicitly deleted
```

Constructors and destructors

Include standard iostream
and string classes

```
#include <iostream>
#include <string>
```

```
using namespace std;
class MyClass {
private:
    string name;
public:
```

Use initialization
list in constructor

```
    MyClass(string name) : name(name) {           // constructor
        cout << "create " << name << endl;
```

```
    }
```

```
    ~MyClass() {
```

Specify cleanup
in destructor

```
        cout << "destroy " << name << endl;
```

```
    }
```

```
};
```

Automatic and dynamic destruction

```
MyClass& start() {                                     // returns a reference
    MyClass a("a");                                   // automatic
    MyClass *b = new MyClass("b"); // dynamic
    return *b;                                         // returns a reference
(!) to b
}                                                       // a goes out of
scope
```

```
void finish(MyClass& b) {
    delete &b;                                         // need pointer to b
}
```

```
#include "MyClass.h"
using namespace std;
int main (int argc, char **argv) {
    MyClass aClass("d");
    finish(start());
    return 0;
}
```

create d
create a
create b
destroy a
destroy b
destroy d



Special Member Functions

- Constructor:
 - Public function member
 - called when a new object is created (instantiated).
 - Initialize data members.
 - Same name as class
 - No return type
 - Several constructors
 - Function overloading



```
class Circle
```

```
{
```

```
    private:
```

```
        double radius;
```

```
    public:
```

```
        Circle() { radius = 0.0;}
```

```
        Circle(int r);
```

```
        void setRadius(double r){radius = r;}
```

```
        double getDiameter(){ return radius *2;}
```

```
        double getArea();
```

```
        double getCircumference();
```

```
};
```

```
Circle::Circle(int r)
```

```
{
```

```
    radius = r;
```

```
}
```

```
double Circle::getArea()
```

```
{
```

```
    return radius * radius * (22.0/7);
```

```
}
```

```
double Circle:: getCircumference()
```

```
{
```

```
    return 2 * radius * (22.0/7);
```

```
}
```



Returning Objects from Functions

```
class Distance
```

```
{
```

```
private:
```

```
    int feet;
```

```
    float inches;
```

```
public:
```

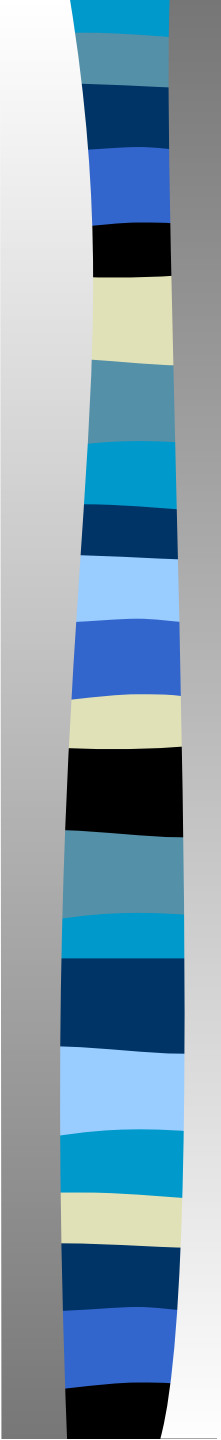
```
    Distance()                // constructor
```

```
{
```

```
    feet = 0;
```

```
    inches = 0.0;
```

```
}
```



```
Distance(int f, float i)           // parameterized constructor
{
    feet = f;
    inches =i;
}

void getdist()                     // get length from user
{
    cout << "Enter Feet: " ;
    cin >> feet;
    cout<< "Enter Inches: ";
    cin>> inches;
}

void showdist()                    // display distance
{
    cout << feet << "\"" << inches << "\"" <<endl;
}
```




```
//add this distance to d2, return sum
```

```
Distance add_dist(Distance d2)
```

```
{
```

```
    Distance temp;
```

```
    temp.inches = inches + d2.inches;
```

```
    if(temp.inches >=12.0)
```

```
    {
```

```
        temp.inches -= 12.0;
```

```
        temp.feet = 1;
```

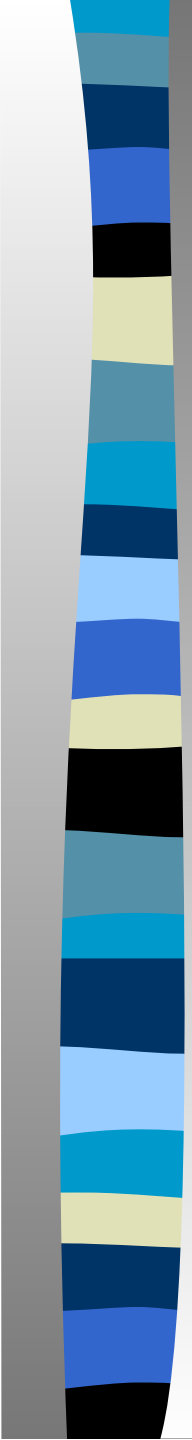
```
    }
```

```
    temp.feet += feet + d2.feet;
```

```
    return temp;
```

```
}
```

```
};    // class ends here
```



```
int main()
{
    Distance dist1, dist3;
    Distance dist2(11,6.25);
    dist1.getdist();           // get dist1 from user
    dist3 = dist1.add_dist(dist2); // dist3 = dist1 + dist2
    //display all lengths
    dist1.showdist();
    dist2.showdist();
    dist3.showdist();

    return 0;
}
```

A decorative vertical bar on the left side of the slide, featuring a series of horizontal stripes in various shades of blue, teal, yellow, and black, set against a light gray background.

Thank you