



CE142:OBJECT ORIENTED PROGRAMMING WITH C++
December 2018 – May 2019

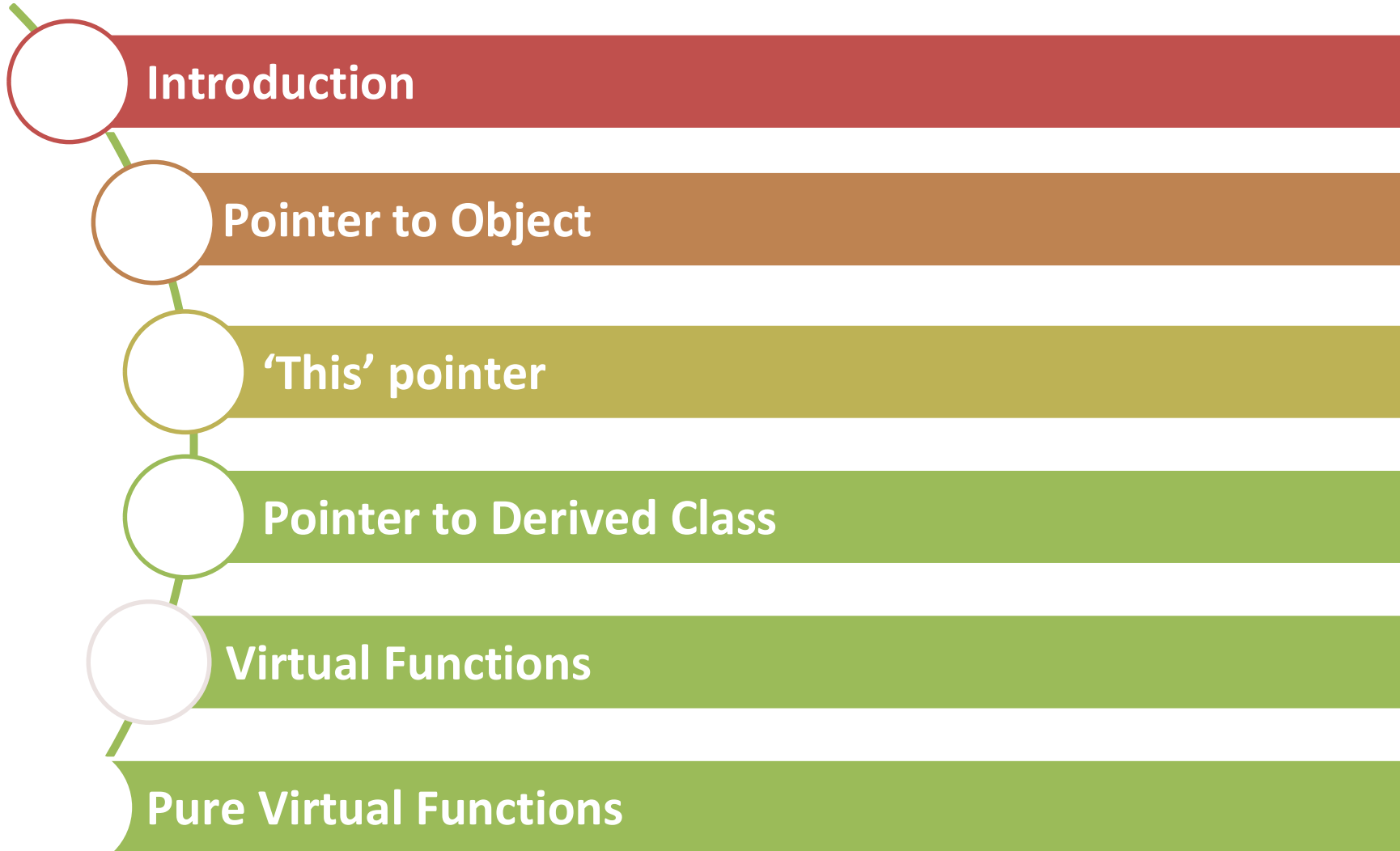
UNIT 10

Pointers and Virtual Functions



Devang Patel Institute of Advance Technology and Research

Content



Introduction

When you declare a variable, the computer associates the variable name with a particular location in memory and stores a value there.

When you refer to the variable by name in your code, the computer must take two steps:

1. Look up the address that the variable name corresponds to
2. Go to that location in memory and retrieve or set the value it contains

C++ allows us to perform either one of these steps independently on a variable with the `&` and `*` operators:

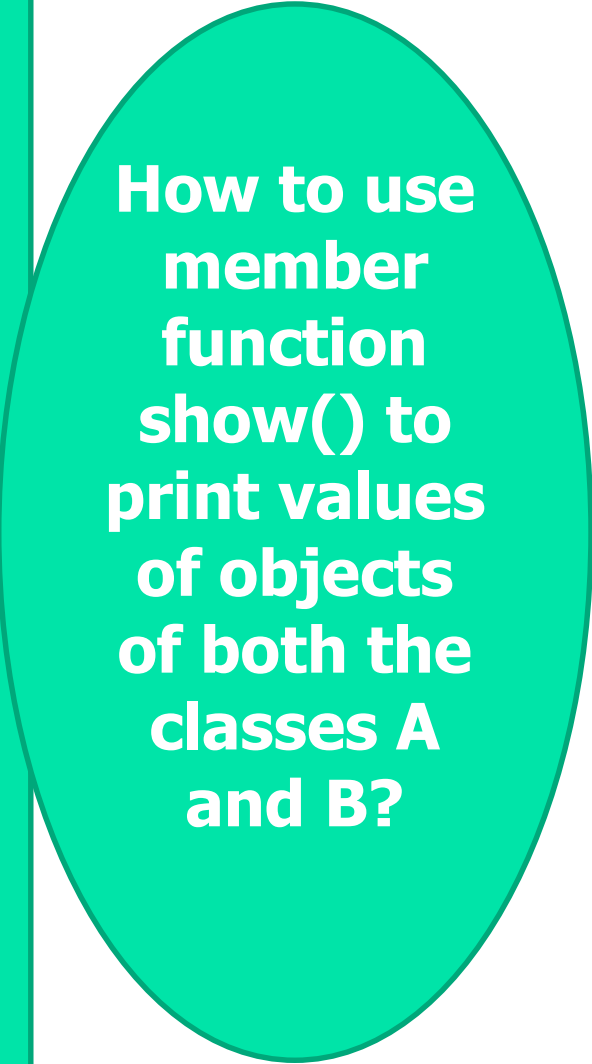
1. `&x` evaluates to the address of `x` in memory.
2. `*(&x)` takes the address of `x` and *dereferences* it – it retrieves the value at that location in memory. `*(&x)` thus evaluates to the same thing as `x`.

Reference: MIT6_096IAP11, January 12, 2011

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms'. We have already seen how the concept of *polymorphism* is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding* or *static binding* or *static linking*. Also known

```
Class A  
{  
int x;  
Public:  
void show()  
  {.....}    // show() in base class  
};
```

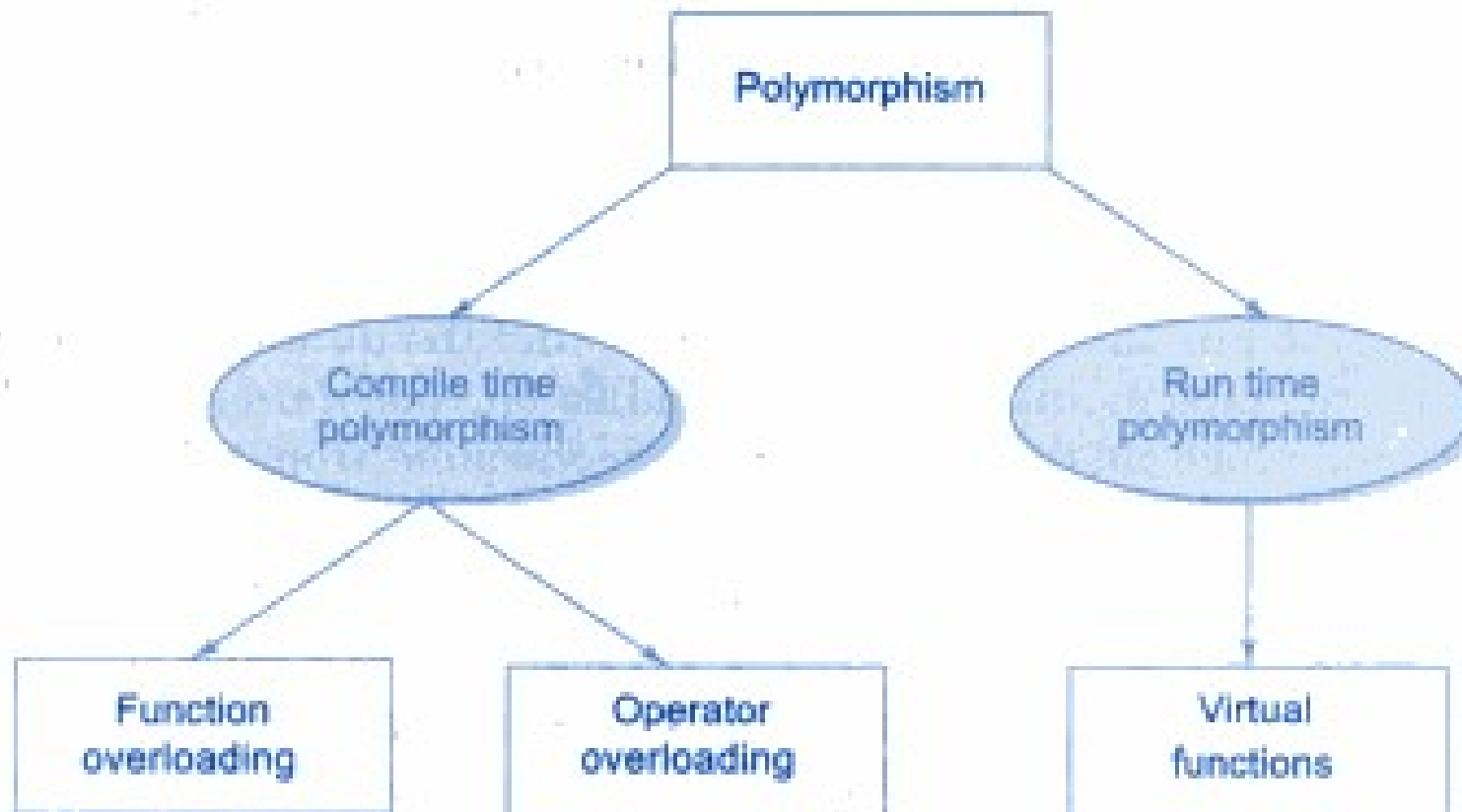
```
Class B :public A  
{  
int y;  
public :  
void show()  
  {.....}    // show() in derived  
};
```



**How to use
member
function
show() to
print values
of objects
of both the
classes A
and B?**

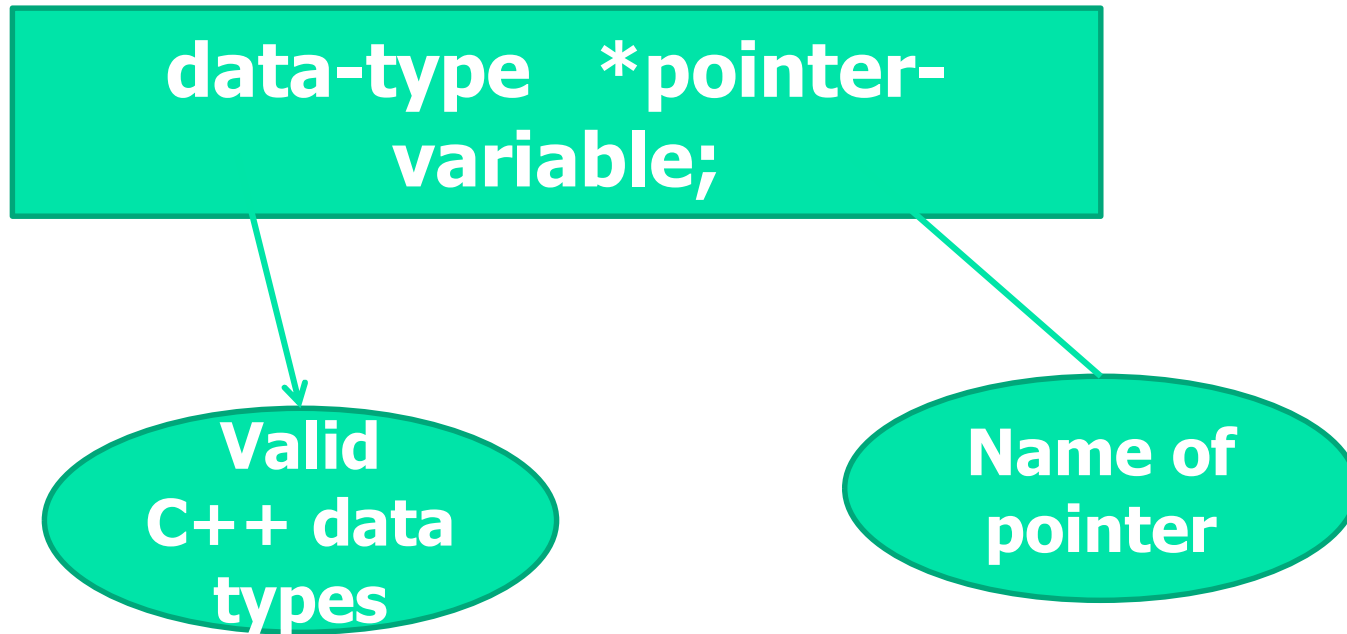
- Solution 1: use class resolution operator
(Static binding)
- Solution 2: appropriate function gets selected while program is running
(dynamic binding / run time polymorphism)

It would be nice if the appropriate member function could be selected while the program is running. This is known as *run time polymorphism*. How could it happen? C++ supports a mechanism known as *virtual function* to achieve run time polymorphism. Please refer



Pointers

- A pointer is a derived data type that refers to another data variable by storing the variables memory address rather than data.
- Declaring and initializing pointers

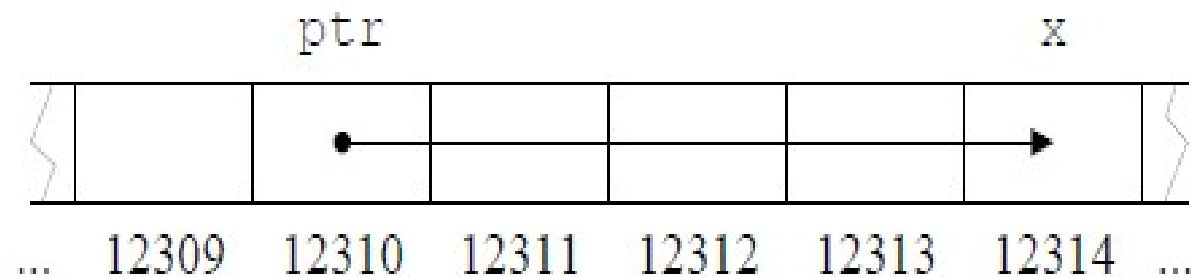


- Pointer is able to point to only one data type at the specific time.

```
int *ptr, a; // declaration  
ptr=&a; // ASSIGNMENT
```

```
int *ptr1,**ptr2 a; // declaration  
ptr1=&a; // Address ASSIGNMENT  
ptr2=&ptr1;
```

- Pointers are just variables storing integers – but those integers happen to be memory addresses, usually addresses of other variables.
- A pointer that stores the address of some variable x is said to point to x. We can access the value of x by dereferencing the pointer using * operator.



- To declare a pointer variable named ptr that points to an integer variable named x:

```
int *ptr = &x;
```

- `int *ptr` declares the pointer to an integer value, which we are initializing to the address of x.
- `cout << *ptr; // Prints the value pointed to by ptr`
- `*ptr = 5; // Sets the value of x`
- Some pointers do not point to valid data; dereferencing such a pointer is a runtime error.
- Any pointer set to 0 is called a null pointer, and since there is no memory location 0, it is an invalid pointer. One should generally check whether a pointer is null before dereferencing it.

EXAMPLE OF USING POINTERS

```
#include <iostream.h>
#include <conio.h>
void main()
{
    int a, *ptr1, **ptr2;
    clrscr();
    ptr1 = &a;
    ptr2=&ptr1;
    cout << "The address of a : " << ptr1 << "\n";
    cout << "The address of ptr1 : " << ptr2;
    cout << "\n\n";
    cout << "After incrementing the address values:\n\n";
    ptr1+=2;
    cout << "The address of a : " << ptr1 << "\n";
    ptr2+=2;
    cout << "The address of ptr1 : " << ptr2 << "\n";
}
```

The address of a : 0x8fb6fff4
The address of ptr1: 0x8fb6fff2
After incrementing the address values:
The address of a : 0x8fb6fff8
The address of a : 0x8fb6fff6

example

```
{
    int a=10, *ptr;
    ptr = &a;
    clrscr();
    cout << "The value of a is : " << a;
    cout << "\n\n";
    *ptr=(*ptr)/2;
    cout << "The value of a is : " << (*ptr);
    cout << "\n\n";
}
```

The value of a is : 10

The value of a is : 5

Pointer Arithmetic

- Pointer arithmetic is a way of using subtraction and addition of pointers to move around between locations in memory, typically between array elements.
- The name of an array is actually a pointer to the first element in the array. Writing `myArray[3]` tells the compiler to return the element that is 3 away from the starting element of `myArray`.
- Adding an integer `n` to a pointer produces a new pointer pointing to `n` positions further down in memory.

Pointer Arithmetic

```
long arr [] = {6,0,1,2};
```

```
long *ptr = arr;
```

```
ptr ++;
```

```
ptr2 = arr +3;
```

- When we add 1 to ptr, we don't just want to move to the next byte in memory, since each array element takes up multiple bytes; we want to move to the next element in the array. The C++ compiler automatically takes care of this, using the appropriate step size for adding to and subtracting from pointers.
- Similarly, we can add/subtract two pointers: ptr2 - ptr gives the number of array elements between ptr2 and ptr.

```
void main()
{
    int i=0;
    char *ptr[10] = {
        "books",
        "television",
        "computer",
        "sports"
    };
    char str[25];
    clrscr();
    cout << "\n\n\nEnter your favorite leisure pursuit: " ;
    cin >> str;
```



```
for(i=0; i<4; i++)  
{  
    if(!strcmp(str, *ptr[i]))  
    {  
        cout << "\n\nYour favorite pursuit " << " is available here"  
        << endl;  
        break;  
    }  
    if(i==4)  
        cout << "\n\nYour favorite " << " not available here" << endl;  
    getch();  
}
```

Pointers and strings

```
char num[]="one";  
const char *numptr= "one";
```

- Num contains 'o','n','e','\0'
- Numptr points to 'o'

Pointers to functions

- Used in C++ for dynamic binding and event driven applications.
- Pointer function is known as call back function
- Function pointers can refer to a function.
- Using function pointers we can select functions dynamically at run time.
- Functions can also be passed as arguments.

- C++ provides two types of function pointers:

- Pointers to static member functions,
- Pointers to non-static member functions (requires hidden arguments)

- `data_type(*function_name)();`

```
int (*num_function(int x));
```

```
#include <iostream.h>

typedef void (*FunPtr)(int, int);

void Add(int i, int j)
{
    cout << i << " + " << j << " = " << i + j;
}

void Subtract(int i, int j)
{
    cout << i << " - " << j << " = " << i - j;
}
```

```
void main()  
{  
    FunPtr ptr;  
    ptr = &Add;  
    ptr(1,2);  
    cout << endl;  
    ptr = &Subtract;  
    ptr(3,2);  
}
```

$$1 + 2 = 3$$

$$3 - 2 = 1$$

Pointer to Objects

- We have already seen how we can point pointers to class members, similarly ***Pointers can also point to class objects***. Such pointers can **also be called** as **“Pointer to class”**.
- A pointer pointing to a class object must be of class *type*.
- For example,

```
class car{  
    int price;  
    char colour;  
}c_obj1;  
car *ptr;  
ptr = &c_obj1;
```

```

class Bill
{
    int qty;
    float price;
    float amount;
    public :
    void getdata (int a, float b, float c)
    {
        qty=a;
        price=b;
        amount=c;
    }
    void show()
    {
        cout<<"Quantity : " <<qty <<"\n";
        cout<<"Price : " <<price <<"\n";
        cout<<"Amount : " <<amount <<"\n";
    }
};

int main()
{
    clrscr();
    Bill s;
    Bill *ptr =&s;
    ptr->getdata(45,10.25,45*10.25);
    (*ptr).show();
    return 0;
}

```

The -> and . operators are used to refer to the member functions of the class Bill.

The () are necessary in **(*ptr).show()**; because . has higher precedence than *

(*ptr).show(); is the same as saying **s.show()**;

What will be the output?


```

class Bill
{
    int qty;
    float price;
    float amount;
    public :
    void getdata (int a, float b, float c)
    {
        qty=a;
        price=b;
        amount=c;
    }
    void show()
    {
        cout<<"Quantity : " <<qty <<"\n";
        cout<<"Price : " <<price <<"\n";
        cout<<"Amount : " <<amount <<"\n";
    }
};

int main()
{
    clrscr();
    Bill s;
    Bill *ptr =&s;
    ptr->getdata(45,10.25,45*10.25);
    (*ptr).show();
    return 0;
}

```

What will be the output?

The output will be:

```

Quantity : 45
Price : 10.25
Amount : 461.25

```

- We can also create class objects using pointers and the **new** operator.
- `Bill *ptr = new Bill;`
- We can also create an ***array of objects*** using pointers.
- `Bill *ptr = new Bill[10];`
- To create an ***array of pointers***.
- `Bill *ptr[10];` each pointer from this array of pointers can now point to a different object of class Bill.

‘this’ Pointer

- 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 means that this unique pointer is automatically passed to a member function as an argument when the function is called.
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

```

class Box {
public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
    }
    double Volume() {
        return length * breadth * height;
    }
    int compare(Box box) {
        return this->Volume() > box.Volume();
    }

private:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};

int main(void) {
    Box Box1(3.3, 1.2, 1.5);    // Declare box1
    Box Box2(8.5, 6.0, 2.0);    // Declare box2

    if(Box1.compare(Box2)) {
        cout << "Box2 is smaller than Box1" << endl;
    } else {
        cout << "Box2 is equal to or larger than Box1" << endl;
    }

    return 0;
}

```

What will be the output?

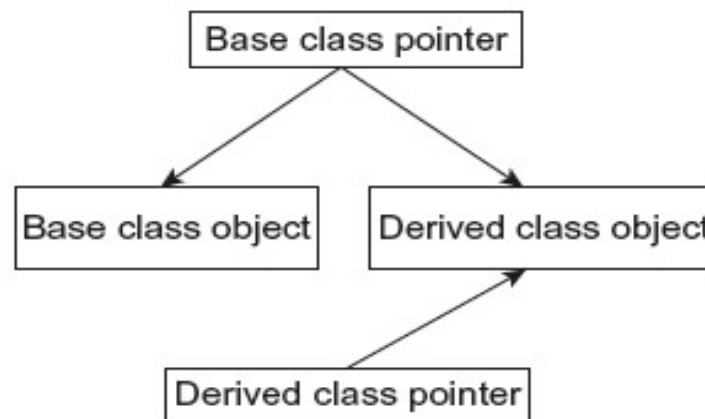
On Next Slide...

The output is...

```
Constructor called.  
Constructor called.  
Box2 is equal to or larger than Box1
```

Pointers to Derived Classes

- It is possible to declare a pointer that points to the base class (*base class object*) as well as the derived class (*derived class object*).
- One pointer can point to different classes.
- For example, X is a base class and Y is a derived class then, a pointer to X can also point to Y.
- **BUT** a pointer to Y *cannot* point to X.



- If B is the base class and D is the derived class, then,

B *cptr; //**pointer to class B type object**

B b; //**base object**

D d; //**derived object**

cptr = &b; //**cptr is pointing to base object b**

cptr = &d; //**cptr is pointing to derived object d**

- cptr can be used to access members of Base Class B including *member variables* and *member functions*.
- After cptr = &d, cptr **cannot** be used to access all members of Derived Class D. Only those members can be accessed which have been derived from Base Class B.

- In case that a member (variable or function) in a derived class D has the same name as a member in the base class B, cptr that is pointing to D will always access the Base Class member with the same name.
- This can be overcome by typecasting cptr as Derived class pointer.

For example,

```
cptr=&b;
```

```
cptr->show(); //calls show() function from Base Class B
```

```
cptr=&d;
```

```
cptr->show(); //still calls show() function from Base Class
```

```
((D*)cptr)->show(); //now calls show() function from Derived Class D  
after typecasting.
```

NOTE: show must be defined in B as well as D with separate definitions (overloaded).

Virtual Functions

- A virtual function is a member function which is declared within base class and is re-defined (Overridden) by derived class.
- They are mainly used to achieve **Runtime polymorphism** i.e. the resolving of function call is done at Run-time.
- Functions are declared with a **virtual** keyword in base class.

```

#include<iostream>
using namespace std;

class base
{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}

```

**WHAT WILL
BE THE
OUTPUT?**

```

#include<iostream>
using namespace std;

class base
{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}

```

ANS:

**PRINT
DERIVED
CLASS**

**SHOW BASE
CLASS**

**Explanation
in next slide**

Explanation

- A base class pointer can point to the objects of base class as well as to the objects of derived class.
- Runtime polymorphism is achieved only through a pointer (or reference) of base class type.
- An Early binding(Compile time) is done according to the type of pointer.
- Late binding(Runtime) is done in accordance with the content of pointer.
- print() function is declared with virtual keyword so it will be binded at run-time and show() is non-virtual so it will be binded during compile time

Pure Virtual Functions

- Sometimes implementation of all functions cannot be provided in a base class because we want different implementations of a function for different derived classes.
- A **Pure virtual function** (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it.
- In the below example, the member function "features" is a pure virtual function

```
class Weapon
{
    public:
        virtual void features() = 0;
};
```

Abstract Class

- Abstract Class is a class which contains at least one Pure Virtual function in it.
- Abstract classes are used to provide an Interface for its sub classes.
- Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.
- Abstract class cannot be instantiated (i.e. objects of abstract class can not be created), but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.

```

//Abstract base class
class Base
{
    public:
    virtual void show() = 0;    // Pure Virtual Function
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Implementation of Virtual Function in Derived class\n";
    }
};

int main()
{
    Base obj;    //Compile Time Error
    Base *b;
    Derived d;
    b = &d;
    b->show();
}

```

Output?

Presentation Prepared By:



Mr. Phenil Buch

Contact us:

phenilbuch.ce@charusat.ac.in
dweepnagarg.ce@charusat.ac.in
parthgoel.ce@charusat.ac.in
hardikjayswal.it@charusat.ac.in
dipakramoliya.ce@charusat.ac.in
krishnapatel.ce@charusat.ac.in
khushipatel.ce@charusat.ac.in

Subject Teachers:



Ms. Dweepna Garg
Subject Coordinator



Mr. Parth Goel
<https://parthgoelblog.wordpress.com>



Mr. Hardik Jayswal



Ms. Khushi Patel

Thank you!

