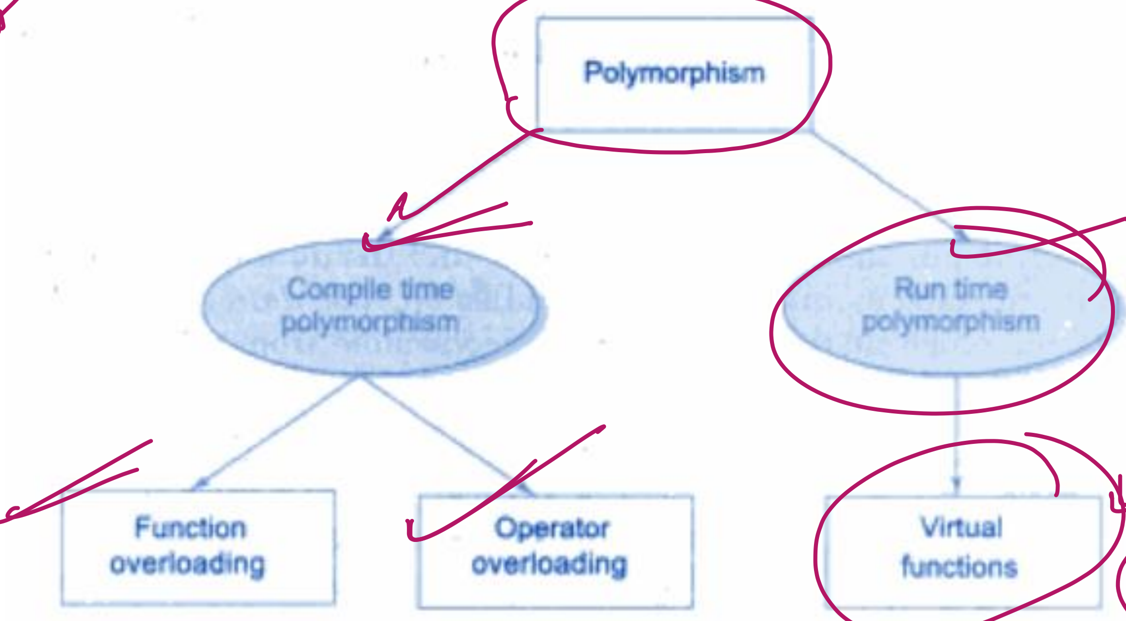
POINTERS, VIRTUAL FUNCTION and POLYMORPHISM

Bu da / Ra

Polymorphism -> 'one name, multiple from'

- Overloading/static binding/early binding/static linking/ **Compile time polymorphism**
- Dynamic binding/ **Run time polymorphism.** (Achieved using overriding only)



fun

What is overriding then? -> Function overriding means creating a newer version of the parent class function in the child class.

func()

POINTERS

Stores memory address (of a particular datatype), a different approach to call variable;

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

void *ptr;

ptr = &a
ptr = &c
&d

Smart pointer/ Generic pointer

We can also use *void pointers*, known as generic pointers, which refer to variables of any data type. Before using void pointers, we must type cast the variables to the specific data types that they point to.

Cannot be dereferenced

cout<<*ptr

Operations on pointer

A pointer can be incremented (++) (or) decremented (– –)
Any integer can be added to or subtracted from a pointer
One pointer can be subtracted from another

cout<<
(int)ptr

Pointer → stores memory loc

# If incremented -> points to next location, similarly for decrementing

# If we add or subtract from pointer, they will point to corresponding memory location

char 1bit
int 4bit

DIFFERENCES BETWEEN ARRAYS AND POINTERS

- Array refers to block of memory, Pointer do not refer to any section of memory
- Memory address of array cannot be changed, but in pointers it can be

int a[]

a

IMPORTANT:

There is no error checking of array bounds in C++. Suppose we declare an array of size 25. The compiler issues no warnings if we attempt to access 26th location. It is the programmer's task to check the array limits.

garbage

ALSO,

We can declare the pointers to arrays as follows:

```
int *nptr;
nptr=number[0];
```

int number[100];

Or

```
nptr=number;
```

ptr

Here, **nptr** points to the first element of the integer array, number[0].

POINTERS AND STRINGS → array of char

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

n e \0

The first declaration creates an array of four characters, which contains the characters, 'o','n','e','\0', whereas the second declaration generates a pointer variable, which points to the first character, i.e. 'o' of the string. There is numerous string handling functions available in C++. All of these functions are available in the header file <cstring>.

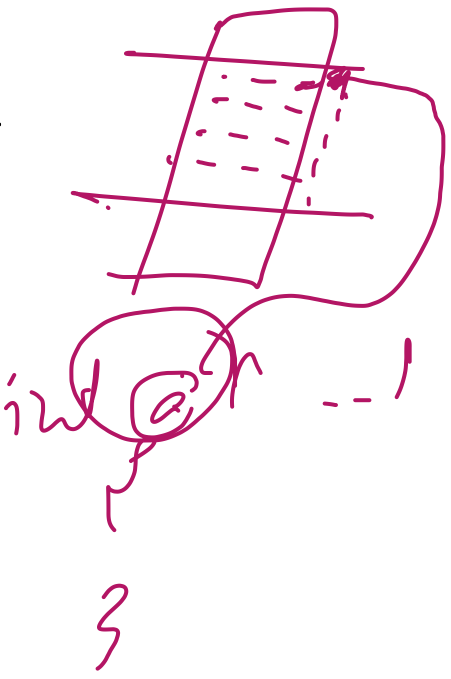
POINTERS TO FUNCTION (Also known as CALLBACK FUNCTION)

for more understanding watch this NESO video: https://youtu.be/BRsv3ZXoHto

We use function pointers to refer to a function, just as we do for data

As function is a set of instruction, stored in memory

And function name is referring to first memory address of function.



SYNTAX:

```
        data_type (*pointer_name)(argument of function)
```

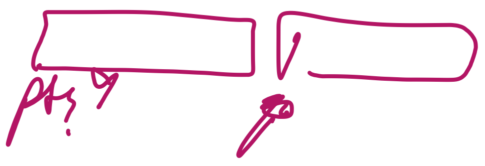
e.g.:

```
        int add(int a, int b) { return a+b; }
        int sub(int a, int b) { return a-b; }
        int main()
        {
                Int (*ptr)(int,int);
                ptr = add;
                cout<<ptr(2,3); // result is 5
                ptr = sub;
                cout<<ptr(2,3); // result is -1
        }
```

POINTERS TO OBJECTS

Suppose we have a class **item**

```
        item x;
        item *ptr;
```



Then **ptr** is now pointer of datatype **item**

- ➔ Suppose class have get function to print data members

get()

If we do ptr = &x;
Hence, the ptr will now point to memory address where x is stored.

We can use get function as follows

```
x.get();
(*ptr).get()     // first derefencing the pointer, and then calling the function
ptr->get()       // this is short hand notation for the above line
```

THIS POINTER

obj → member func

Suppose we have object **A** of class **item**.

Get function in class is generally defined as ->

```
void get()
{
        cout<<data<<"\n";
}
```

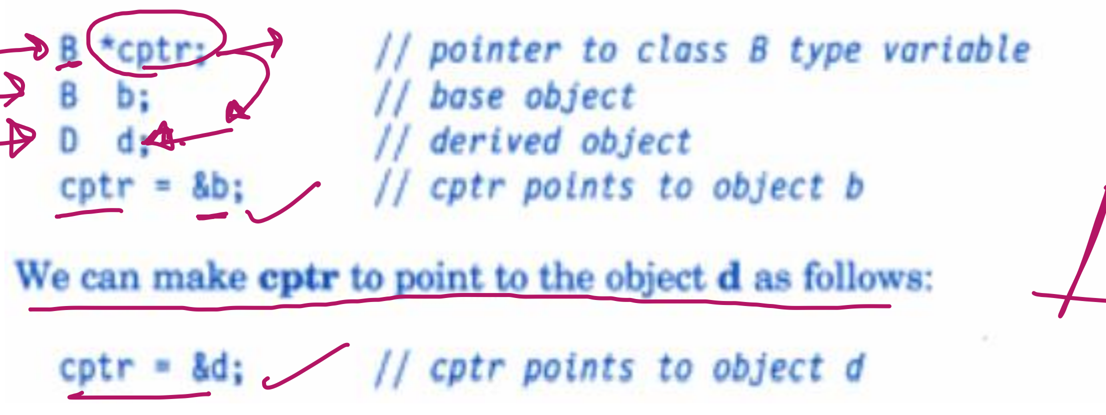
We can use **this** keyword also as ⇐ just works same to. address of calling obj

```
void get()
{
        cout<<this->data<<"\n";
}
```

What happened here is, whichever object invokes the member function or data member, **this** keyword stores its address.

Hence, when **A.get()** is called, **this** stored the address of object **A**.

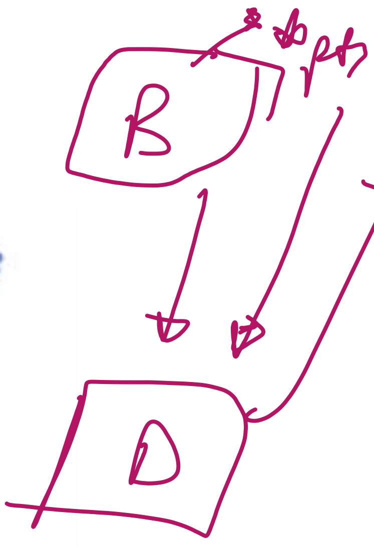
POINTERS TO DERIVED CLASS



```
B *cptr;          // pointer to class B type variable
B  b;             // base object
D  d;             // derived object
cptr = &b;        // cptr points to object b
```

We can make **cptr** to point to the object **d** as follows:

```
cptr = &d;        // cptr points to object d
```



As **cptr** is pointer of base class, but storing address of derived class, hence **cptr** can access only data members or member functions inherited. Not that who are originally from derived class.

D → fun ("Derived")
fun ("Base")

B* ptr ;   D d ;

ptr = & d

ptr → fun →

Because a derived class includes everything that is in the base class. But a base class does not include everything that is in the derived class.

Type casting a base class to a derived class is not recommended: What happens if you try to access members that are not part of the base class?

VIRTUAL FUNCTION

Suppose this -> We have a base class pointer and it points to derived class! Possible right.

Now if we call a function which is derived from base class, using this pointer. It stills works as defined in Base class, The compiler simply ignores the contents of pointer and chooses the member function that matches the type of pointer (i.e. base class)

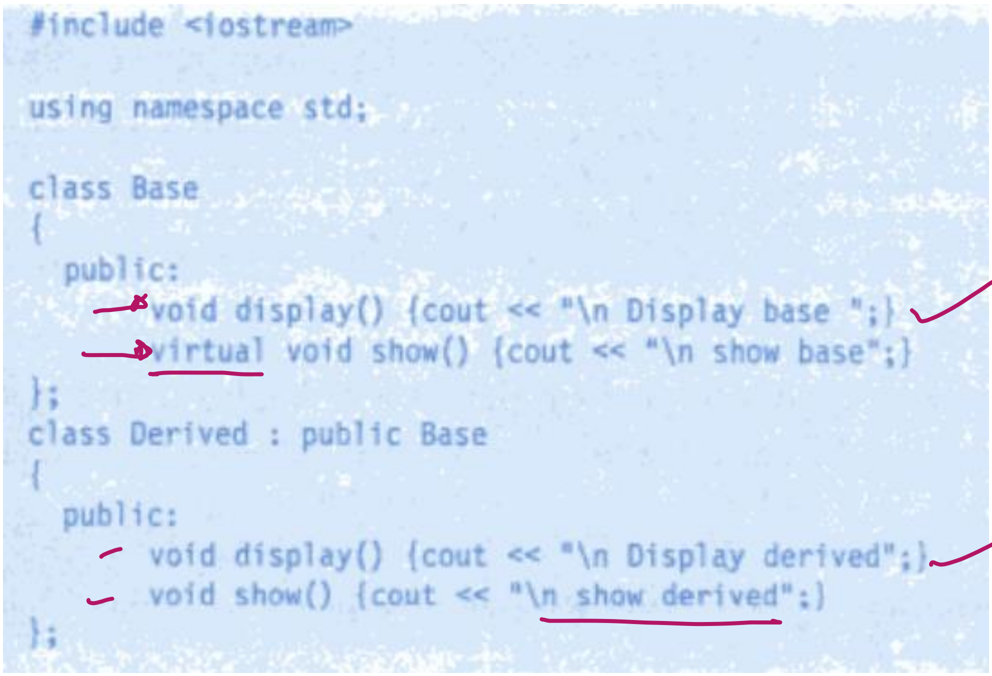
Then how we could achieve polymorphism? Using **Virtual Function**

Hence to make function on bases of content of pointer, not type of it. We use keyword **virtual** in front of function definition in BASE CLASS

Hence compiler will decide which function to call on basis of content of pointer.

Hence to Achieve runtime polymorphism -> Compiler should get to know at compile time what are the contents of pointer and then call the function

We use pointers not objects for this. (Pointer of base class)



```cpp
#include <iostream>

using namespace std;

class Base
{
  public:
    void display() {cout << "\n Display base ";}
    virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
  public:
    void display() {cout << "\n Display derived";}
    void show() {cout << "\n show derived";}
};
```

```
Base B;
Derived D;
Base *bptr;

cout << "\n bptr points to Base \n";
bptr = &B;
bptr -> display();     // calls Base version
bptr -> show();        // calls Base version

cout << "\n\n bptr points to Derived\n";
bptr = &D;
bptr -> display();     // calls Base version
bptr -> show();        // calls Derived version
```

RULES OF VIRTUAL FUNCTION

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

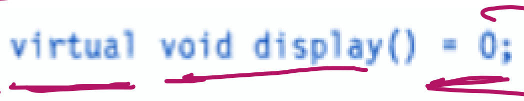
Abstract Class

$\hookrightarrow$ objects

PURE VIRTUAL FUNCTION

Well generally we create a skeleton function which gets inherited in derived class and we redeclare it in derived class.

So we can create this skeleton function/ do-nothing function in base class as ->



```
virtual void display() = 0;
```

Such functions are called **pure virtual function,** here we have no definition of this class, we need to redeclare this function in derived class.

Such Base classes containing **Pure Virtual Functions** are called **Abstract Class**. Hence we cannot objects of such classes.