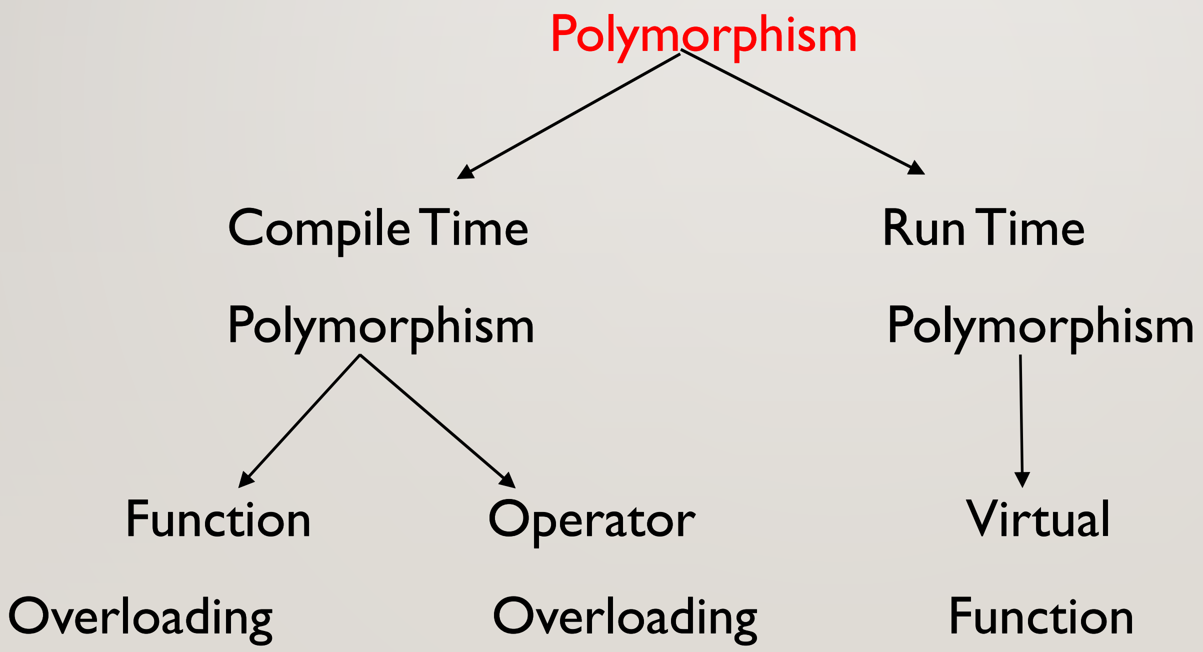
CHAPTER 7: POLYMORPHISM AND DYNAMIC BINDING.

(4 HOURS)

POLYMORPHISM AND ITS TYPES:

Poly->Many(Multiple).

Morphism->Forms.



Polymorphism is a the ability of an object or reference to take many different forms at different instances.

Real Life Example:

A Person behaves as :

- ✓ passenger in bus.
- ✓ Student in school.
- ✓ Son/Daughter in house.

EARLY BINDING AND LATE BINDING:

To understand **virtual function**, first we need to know about early and late binding.

In Compile time polymorphism , object is bound to the function call at the compile time itself . This is called as **early binding or static binding**. It is default binding and done by the compiler.

In Run time polymorphism, object is bound to the function call only at the run time . This is called as **late binding or dynamic binding**. In late binding , the compiler matches the function call with the correct function definition at runtime. Late binding can be performed by using virtual function.

BASE CLASS POINTER:

- Base class pointer can point to the object of any of its descendent class.
- But its converse is not true.

Virtual Function:

- Virtual means existing in appearance but not in reality.
- When virtual functions are used , a program that appears to be calling a function of one class may in reality be calling a function of a different class.
- When we use the same function name in both base class and derived classes , the function in base class is declared as virtual.
- When member function of base class is declared with the keyword virtual, then it is known as virtual function.
- C++ determines which function to use at run time based on the type of object pointed by the base pointer , rather than the type of pointer.
- Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Syntax Of Virtual Function:

```
 class classname{
 private:
//……
  public:
   virtual return_type function_name (args..)
{
//body of virtual function
}
//……
};
```

NEED OF VIRTUAL FUNCTION:

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve ***Runtime polymorphism***.
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules of Virtual Function:

- They must be declared in public section of class.
- Virtual functions cannot be **static**.
- Virtual functions should be accessed using **pointer or reference of base class type** to achieve run time polymorphism.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class. It is not compulsory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

- **NOTE:** If we have created virtual function in base class and it is being overrided in derived class then we don't need virtual keyword in derived class, functions are automatically considered as virtual functions in derived class.

<u>To Do:</u>

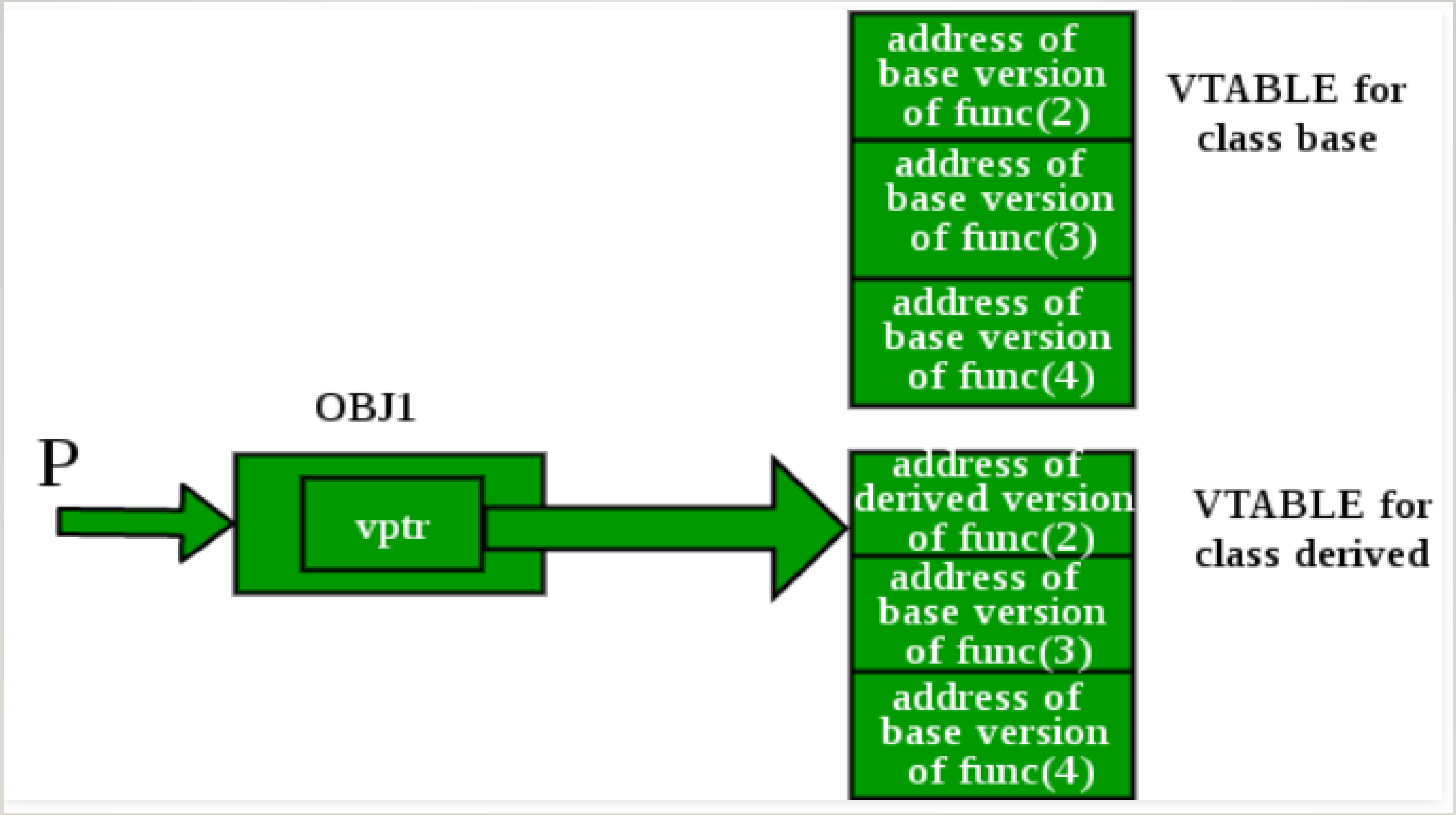
Pointer To Derived Class.

Array Of Pointer To Base Class.

WORKING CONCEPT OF VIRTUAL FUNCTION (CONCEPT OF VPTR AND VTABLE):

❖ **If a class contains a virtual function then compiler itself does two things:**

- If object of that class is created then a **virtual pointer(VPTR)** is inserted as a data member of the class to point to VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
- Irrespective of object is created or not, **a static array of function pointer called VTABLE** where each cell contains the address of each virtual function contained in that class.


address of base version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class base
OBJ1
P
vptr
address of derived version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class derived

PURE VIRTUAL FUNCTION:

- A pure virtual function (or abstract function) in C++ is a **virtual function** for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.
- A do nothing function is pure virtual function.
- Syntax:

```
 class class_name {
   public:
   virtual return_type function_name()=0;     };
```

```
e.g.  class person {
   public:
         virtual void eat()=0;
};
```

ABSTRACT BASE CLASS OR ABSTRACT CLASS:

- A class containing at least one pure virtual function is an abstract class.
- We can not instantiate abstract class i.e. object of abstract class cannot be made.
- We can have pointers and references of abstract class type.
- If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
- An abstract class can have constructors.

IMPORTANT TO KNOW:

- The purpose of an **abstract class** is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**.
- Classes that can be used to instantiate objects are called **concrete classes**.
- Classes that can have at least one virtual function on them are **polymorphic classes**.
- We can call **private** function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

VIRTUAL DESTRUCTOR:

- When base class and derived class have destructors then to clean up object, both base class and derived class destructors must be called.
- When a derived class object pointed by base class pointer is deleted using delete operator with base class pointer it calls the destructor of base class only but not of derived class.
- Hence in order to call both, the destructor of base class must made virtual.
- Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destruction of the object when the program exits.
- Constructors cannot be virtual but destructor can be virtual.

UP CASTING AND DOWN CASTING:

- **Upcasting** is converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as though it were its base type. It is always allowed for **public** inheritance, without an explicit type cast. This is a result of the **is-a** relationship between the base and derived classes.

  Parent *p = &child; //compile//no error occur.

- The opposite process, converting a base-class pointer (or reference) to a derived-class pointer (or reference) is called **downcasting**. Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the **is-a** relationship is not, in most of the cases, symmetric. A derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

  Child *c= &parent; //won't compile//error: cannot convert from parent * to child *.

REINTERPRET_ CAST OPERATOR:

- reinterpret-cast operator is used to convert one type into a fundamentally

  different type.

  e.g. it can be used to change pointer type object to integer type object and vice versa.

- It can be used for casting inherently incompatible pointer types.
- It can perform dangerous conversion because it can typecast any pointer to any other pointer.
- It is used when you want to work with bits.

Syntax:

```
reinterpret_cast <target_type> (expr);
```

Where, target_type specifies the target type of the cast and expr being cast into new type.

- The result of reinterpret_cast cannot safely be used for anything other than being cast back to its original type.
- We should be very careful when using this cast.
- If we use this type of cast then it becomes non-portable product.

❖ **The reinterpret_cast is used for casts that are not safe:**

- Between integers and pointers
- Between pointers and pointers
- Between function-pointers and function-pointers

Note: The reinterpret_cast converts the object from one type to another type without checking the bit pattern or size of source and destination type . So if the number of bit of source and destination is same then the conversion in vice versa takes place properly. But if there is difference in bit than the conversion in vice versa does not takes properly.

RUN-TIME TYPE INFORMATION (RTTI):

- Run-time type information is one of the feature that is available in a language exhibiting run time polymorphism.
- In C++, RTTI (Run-time type information) is a mechanism that exposes information about an object's data type at runtime and is available only for the classes which have at least one virtual function.
- It allows the type of an object to be determined during program execution.
- The operator dynamic_cast and typeid provides us the run time type information.

DYNAMIC_CAST OPERATOR:

- The dynamic_cast operator is one of the important operator among the new casting operator.
- The dynamic_cast operator performs a runtime cast along with verifying the validity of the cast.
- If the cast is invalid then the cast fails.
- Syntax:

```
dynamic_cast < target-type > (expr) ;
```

Where, target_type specifies the target type into which the cast convert the type to and expr is expression that is being cast into the new type.

- Here the target type must be a pointer or reference type and expression being cast must evaluate to a pointer or reference.
- So , dynamic cast may be used to cast one type of pointer into another or one type of reference to another.
- This operator is actually used to perform cast on polymorphic types.

[Important Points in dynamic cast:]

1. Dynamic cast is used for upcast and downcast .It is mainly used for correct downcast.
2. Needs at least one virtual function in base class.
3. If cast is successful, dynamic cast return a value of type new type.
4. If cast is unsuccessful, and new type is a pointer type, it returns a null pointer of that type.
5. If cast is unsuccessful , and new type is a reference type, it throws an exception that matches a handler of type std::bad_cast.
6. Using this cast has runtime overhead, because it checks objetS types at run time using RTTI.

REAL TIME USE AND FLAWS OF DYNAMIC CAST:

- The dynamic cast operation allows flexibility in the design and use of data management facilities in object-oriented programs.
- Like similar operators in other languages, the C++ dynamic cast operator does not provide the timing guarantees needed for hard real-time embedded systems.
- "The need for **dynamic_cast** generally arises because we want to perform **derived class operation** on a **derived class object**, but we have only a pointer-or reference-to-**base**."

TYPEID OPERATOR:

- Its an operator defined in typeinfo.h header.
- It's a method which returns type_info structure which is used to query information of a type.
- It is used for static type identification.
- It is used to find the dynamic type of a polymorphic object with virtual function.
- The typeid operator returns reference to a standard library class type type_info

  defined in <type_info> .
- It is most commonly used to find the type of an object refered by a reference or a pointer.
- Syntax: typeid (expr);

or, typeid (type_name);

- If the value of pointer or reference operand is NULL(0), typeid throws a bad_ typeid exception.

❖ **The type_info class has the following public members:**

```
bool operator == (const type_info &) const;      //for comparison.
bool operator != (const type_info &) const;      //for comparison.
bool before(const type_info &) const;        //for ordering.
const char * name();      //name of type.//The name() function returns a pointer to the name of the type.
```