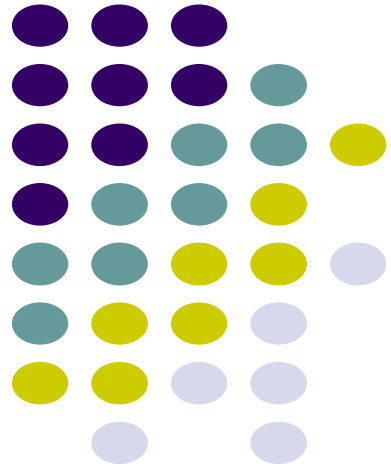
Object Oriented Programming in C++

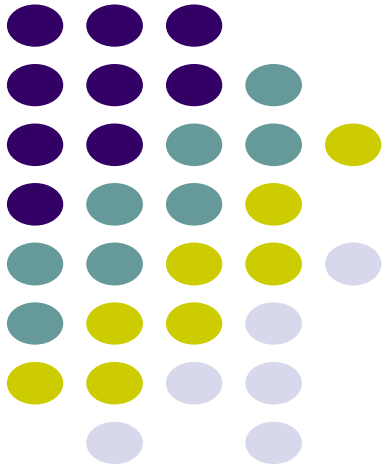
Presented by Bhimashankar T

E-Mail: bhima.t@gmail.com

PhNo:+91 9980156833 / 6363863430

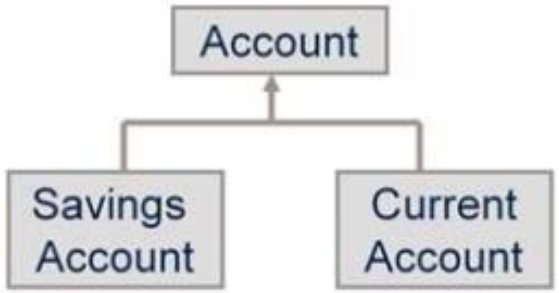


DAY 04

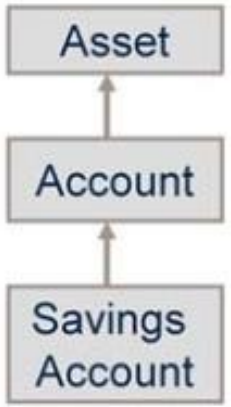


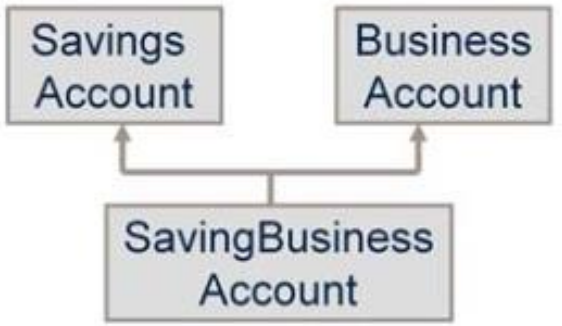
3.1: Object-Oriented Principles

Types of Inheritance Hierarchy



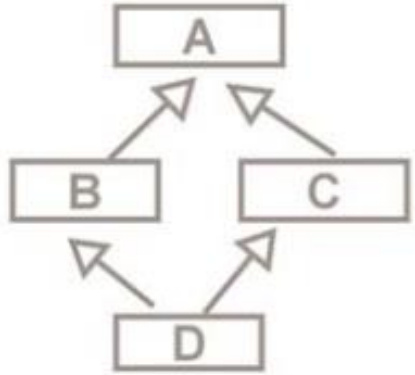
Single-level inheritance





Multiple inheritance

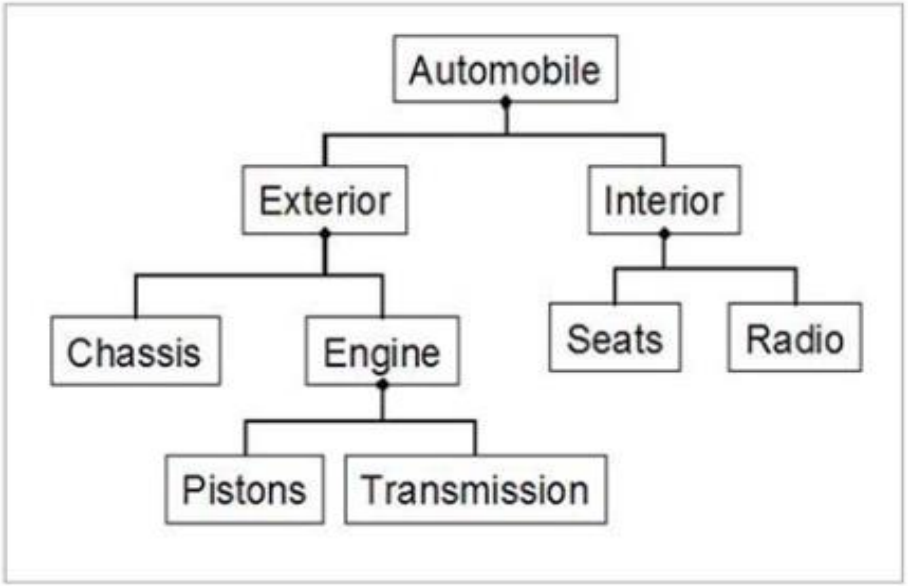
Multiple inheritance challenges: A name conflict introduced by a shared super-class (A) of super-classes (B and C) used with "multiple inheritance".



3.1: Object-Oriented Principles

Object Hierarchy

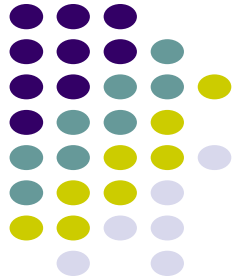
- "Has-a" hierarchy is a relationship where one object "belongs" to (is a part or member of) another object, and behaves according to the rules of ownership.



Note: The container hierarchy (has-a hierarchy) is in contrast to the inheritance hierarchy, i.e., the "generic-specific" levels do not come in here.

A glance at relationships

- The Inheritance or “Is A” Hierarchy leads to Generalization relationship amongst the classes.
- The Object Hierarchy or “Has A” relationship leads to Containment relationship amongst the objects
  - Aggregation and Composition are two forms of containment amongst objects
  - Aggregation is a loosely bound containment. Eg. Library and Books, Department and Employees
  - Composition is tightly bound containment. Eg. Book and Pages



As seen earlier, in an inheritance hierarchy, the super class is the more generic class, and subclasses extend from the generic class to add their specific structure and behaviour. The relationship amongst these classes is a generalization relationship. OO Languages provide specific syntaxes to implement inheritance or the generalization relationship.
Has A or Containment is further of two forms depending on how tight is the binding between the container (“Whole”) and its constituents (“Part”). In the whole-part relationship, if the binding is loose i.e. the contained object can have an independent existence, the objects are said to be in an aggregation relationship. On the other hand, if the constituent and the container are tightly bound (Eg. Body & parts like Heart, Brain..), the objects are said to be in a composition relationship.

A glance at relationships

- Most commonly found relationship between classes is Association
  - Association is the simplest relationship between two classes
  - Association implies that an object of one class can access public members of an object of the other class to which it is associated

The relationship we are most likely to see amongst classes is the Association Relationship. When two classes have an association relationship between them, it would mean that an object of one class can access the public members of the other class with which it is associated. For eg. if a “Sportsman” class is associated with “Charity” class, it means that a Sportsman object can access features such as “View upcoming Charity Events” or “Donate Funds” which are defined within the “Charity” class.

Key Feature – Polymorphism

- It implies One Name, Many Forms.
- It is the ability to hide multiple implementations behind a single interface.
- There are two types of Polymorphism, namelv:
  - Static Polymorphism
  - Dynamic Polymorphism

Polymorphism:
The word Polymorphism is derived from the Greek word "Polymorphous", which literally means "having many forms".
Polymorphism allows different objects to respond to the same message in different ways!
There are two types of polymorphism, namely:

Static (or compile time) polymorphism, and
Dynamic (or run time) polymorphism

Key Feature – Static Polymorphism

- Resolution of the "Form" is at compile time, achieved through overloading.

```
int addInteger(int, int);
float addFloat(float, float);
double addDouble(double, double);
```

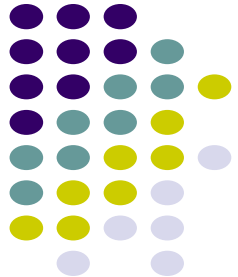
Traditional Approach

Overloaded Functions

```
int addNumber(int, int);
float addNumber(float, float);
double addNumber(double, double);
```

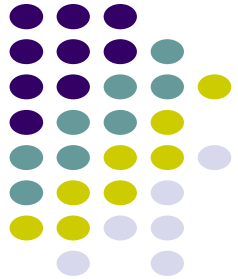
Resolution At Compile Time

Though the name is the same, the right function is called depending on number and/or types of parameters that are there in the function invocation



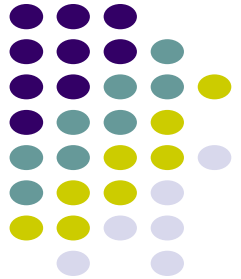
Polymorphism (contd.):
Overloading is when functions having same name but different parameters (types or number of parameters) are written in the code. When Multiple Sort operations are written, each having different parameter types, the right function is called based on the parameter type used to invoke the operation in the code. This can be resolved at compile time itself since the type of parameter is known.

Operator Overloading – walk through the codes



Codes for Operator Overloading (click here)

C++ Operators that cannot be Overloaded



There are four C++ operators that can't be overloaded. They include:

1. :: Scope resolution operator
2. ?: ternary operator.
3. . member selection operator
4. .* member selection through a pointer to function operator

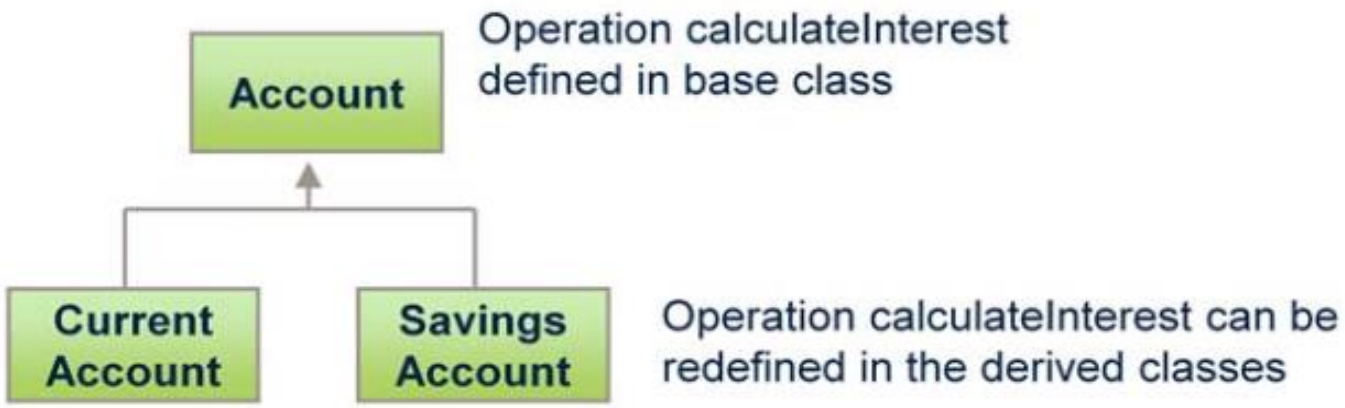
C++ Operators that can be Overloaded

- Following is the list of operators which can be overloaded –

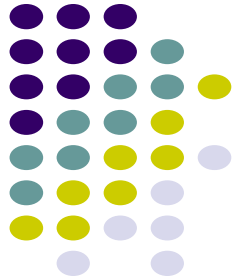
+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Key Feature – Dynamic Polymorphism

- Resolution of the "Form" is at run time, achieved through overriding.



The right operation defined in one of these classes is invoked at Run Time depending on which object is invoking the operation.



Polymorphism (contd.):

On the other hand, the function calculateInterest can be coded across different account classes. At runtime, based on which type of Account object (i.e., object of Current or Savings Account) is invoking the operation, the right operation will be referenced. Overriding is when functions with same signature provide for different implementations across a hierarchy of classes.

As seen in the example here, inheritance hierarchy is required for these objects to exhibit polymorphic behaviour. The classes here are related since they are different types of Accounts, so it is possible to put them together in an inheritance hierarchy. Does that mean that polymorphism is possible only with related classes in an inheritance hierarchy? The answer is No!

We can have unrelated classes participating in polymorphic behavior with the help of the “Interface” concept, which we shall study in a subsequent section.

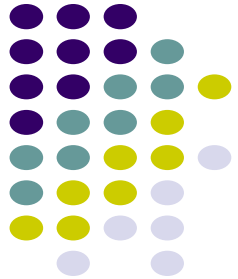
Key Feature – Polymorphism

- Why Polymorphism?
  - It provides flexibility in extending the application.
  - It results in more compact designs and code.

Polymorphism (contd.):
If the banking system needs a new kind of Account, extending without rewriting the original code, then it is possible with the help of polymorphism.
In a Non OO system, we would write code that may look something like this:

```
IF Account is of CurrentAccountType THEN
        calculateInterest_CurrentAccount()
IF Account is of SavingsAccountType THEN
        calculateInterest_SavingAccount()
```

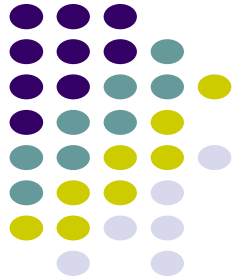
The same in a OO system would be myAccount.calculateInterest(). With object technology, each Account is represented by a class, and each class will know how to calculate interest for its type. The requesting object simply needs to ask the specific object (For example: SavingsAccount) to calculate interest. The requesting object does not need to keep track of three different operation signatures.



Objectives

In this lesson, you will learn to:

- Describe static, or early, or compile-time binding
- Describe dynamic, or runtime, or late binding
- Describe a virtual function
- Employ a virtual function to implement dynamic binding
- Achieve runtime polymorphism through dynamic binding, a base class pointer, and virtual functions

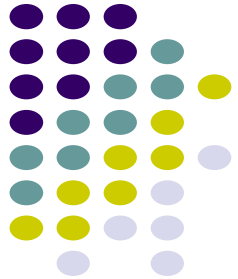


Virtual Functions

A virtual function is a member function that is declared within a base class, and is redefined by a derived class.

To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**.

When a class containing a virtual function is inherited, the derived class redefines or overrides the virtual function to fit its own needs.

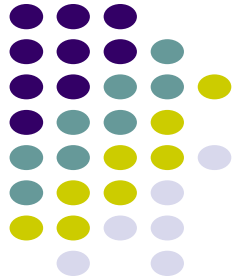


Virtual Functions

Virtual functions implement the “**single method; multiple implementations**” paradigm intrinsic to polymorphism.

The virtual function within the base class defines the form of the interface to the function.

Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method.

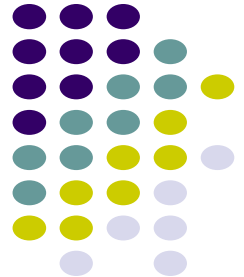


Virtual Functions

When a base class pointer points to a derived class object that contains a virtual function, C++ determines which version of that function to call based upon the type of the object pointed to by the pointer. **And this determination is made at runtime.**

Thus, when different derived class objects are pointed to by the base class pointer at different points of time, different versions of the virtual function are executed.

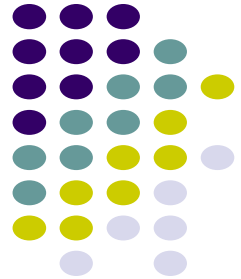
What makes virtual functions important and capable of supporting runtime polymorphism is how they behave when accessed via a pointer. To recapitulate, a base class pointer can be used to point to an object of any class derived from that base.



Virtual Functions

```
#include<iostream>
using namespace std;
class base
 {
   public:
    virtual void vfunc( )
      { cout << “this is base’s vfunc( )\n”; } };

class derived1: public base
 {
   public:
    void vfunc( )
      {  cout << “this is derived1’s vfunc( )\n”; }
};
```

```
class derived2 : public base
 {
   public:
    void vfunc( )
      { cout << “this is derived2’s vfunc( )\n”; }
 };

int main( )
 {
        base *p, b;
        derived1 d1;
        derived2 d2;
        p = &b;
        p->vfunc( ); //access to base’s vfunc( )
        p = &d1;
        p->vfunc( ); // access derived1’s vfunc( )
        p = &d2;
        p->vfunc( ); // access derived2’s vfunc( )
        return 0;
}
```

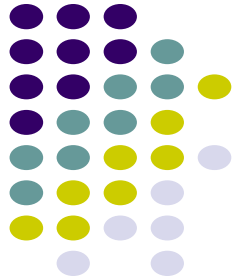


Virtual Functions

The key point here is that the kind of object to which p points to determines which version of vfunc( ) is executed.

Further, this determination is made at runtime, and this process forms the basis of runtime polymorphism.

Although you can call a virtual function in the normal manner by using a object's name and the dot operator, *it is only when access is through a base-class pointer (or reference) that runtime polymorphism is achieved.*

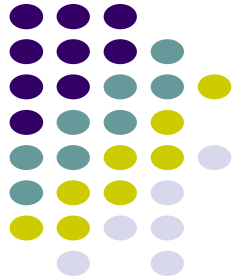


Virtual Functions

A function declared as virtual in the base class, and redefined in the derived classes is the implementation of overridden functions.

The prototype for a redefined or overridden virtual function must exactly match the prototype specified in the base class.

Prototype encompasses not only signature, but also the return data type of a function.



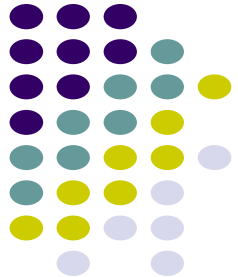
Virtual Functions

At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading. However, this is not the case, and the term overloading is not applied to virtual function redefinition because several differences exist. The most important difference is that the prototype for a redefined or overridden virtual function must exactly match the prototype specified in the base class. This differs from overloading a normal function, in which the number and types of parameters may differ. In fact, when you overload a function, either the number of type of parameters must differ. It is through these differences that C++ can select the correct version of an overloaded function.

However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded, and its virtual nature will be lost.

Another important restriction is that virtual functions must be non-static members of a class.

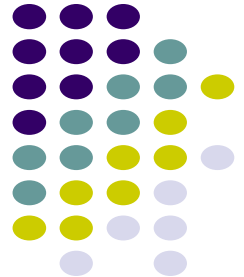
Calling a Virtual Function Through a Base Class Reference



The polymorphic nature of a virtual function is also available when called through a base class reference.

Thus a base class reference can be used to refer to an object of the base class, or any object derived from that base.

When a virtual function is called through a base class reference, the version of the function executed is determined by the object being referred to at the time of call.



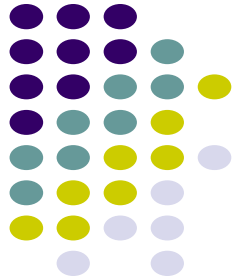
Calling a Virtual Function Through a Base Class Reference

The most common situation in which a virtual function is invoked through a base class reference is when the reference is defined as a function parameter. Consider the following program:

```
#include<iostream>
using namespace std;
class base
 {
   public:
    virtual void vfunc( )
      {  cout << “this is base’s vfunc( )\n”; } };

class derived1: public base
 {
   public:
    void vfunc( )
      { cout << “this is derived1’s vfunc( )\n”; } };
```

Calling a Virtual Function Through a Base Class Reference

```
class derived2 : public base
 {
   public:
    void vfunc( )
      { cout << “this is derived2’s vfunc( )\n”; }
 };
void f(base &r)
 {
                r.vfunc( );
}
int main( )
 {
        base b;
        derived d1;
        derived d2;
        f(b); // pass a base object to f( )
        f(d1); // pass a derived object to f( )
        f(d2); // pass a derived object to f( )
        return 0;
}
```

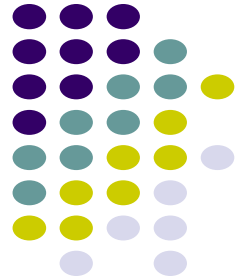


The Virtual Attribute is Inherited

When a virtual function is inherited, its virtual nature is also inherited.

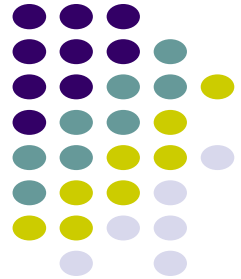
This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.

No matter how many times a virtual function is inherited, it remains virtual.



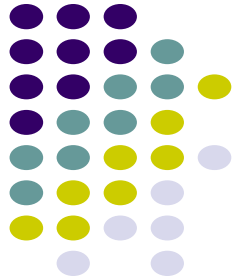
The Virtual Attribute is Inherited

```
#include<iostream>
using namespace std;
class base
 {
   public:
    virtual void vfunc( )
      { cout << “this is base’s vfunc( )\n”; }
};

class derived1: public base
 {
   public:
    void vfunc( )
      { cout << “this is derived1’s vfunc( )\n”; }
};
```



The Virtual Attribute is Inherited

```
class derived2 : public derived1
 {
   public:
    void vfunc( )
      { cout << “this is derived2’s vfunc( )\n”; }
};

int main( )
 {
   base *p, b;
   derived1 d1;
   derived2 d2;
   p = &b;
   p->vfunc( ); //access to base’s vfunc( )
   p = &d1;
   p->vfunc( ); // access derived1’s vfunc( )
   p = &d2;
   p->vfunc( ); // access derived2’s vfunc( )
   return 0;
}
```

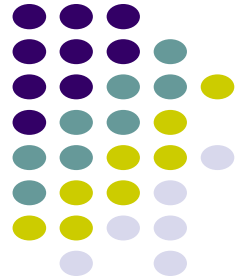


Virtual Functions are Hierarchical

When a function is declared as virtual by a base class, it may be overridden by a derived class. However, the function does not have to be overridden.

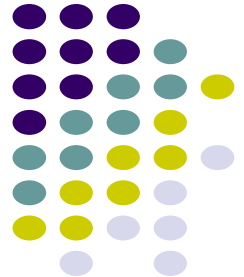
When a derived class fails to override a virtual function, then, when an object of the derived class accesses that function, the function defined by the base class is used.

Consider this program in which class derived2 does not override vfunc( )



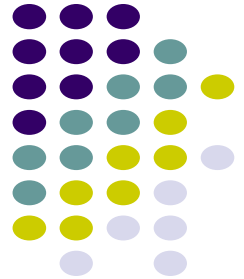
Virtual Functions are Hierarchical

```
#include<iostream>
using namespace std;
class base
 {
   public:
    virtual void vfunc( )
      { cout << “this is base’s vfunc( )\n”; }
};

class derived1: public base
 {
   public:
    void vfunc( )
      {  cout << “this is derived1’s vfunc( )\n”; }
};
```



Virtual Functions are Hierarchical

```
//derived2 inherits virtual function from derived1
class derived2 : public derived1
 {
            // vfunc( ) not overridden by derived2; base's is used
};

int main( )
 { base *p, b;
   derived1 d1;
   derived2 d2;
   p = &b;
   p->vfunc( ); //access to base's vfunc( )
   p = &d1;
   p->vfunc( ); // access derived1's vfunc( )
   p = &d2;
   p->vfunc( ); // access base's vfunc( ) since derived2 does not override vfunc( )
   return 0;
}
```



Virtual Functions are Hierarchical

The preceding program displays the following output:

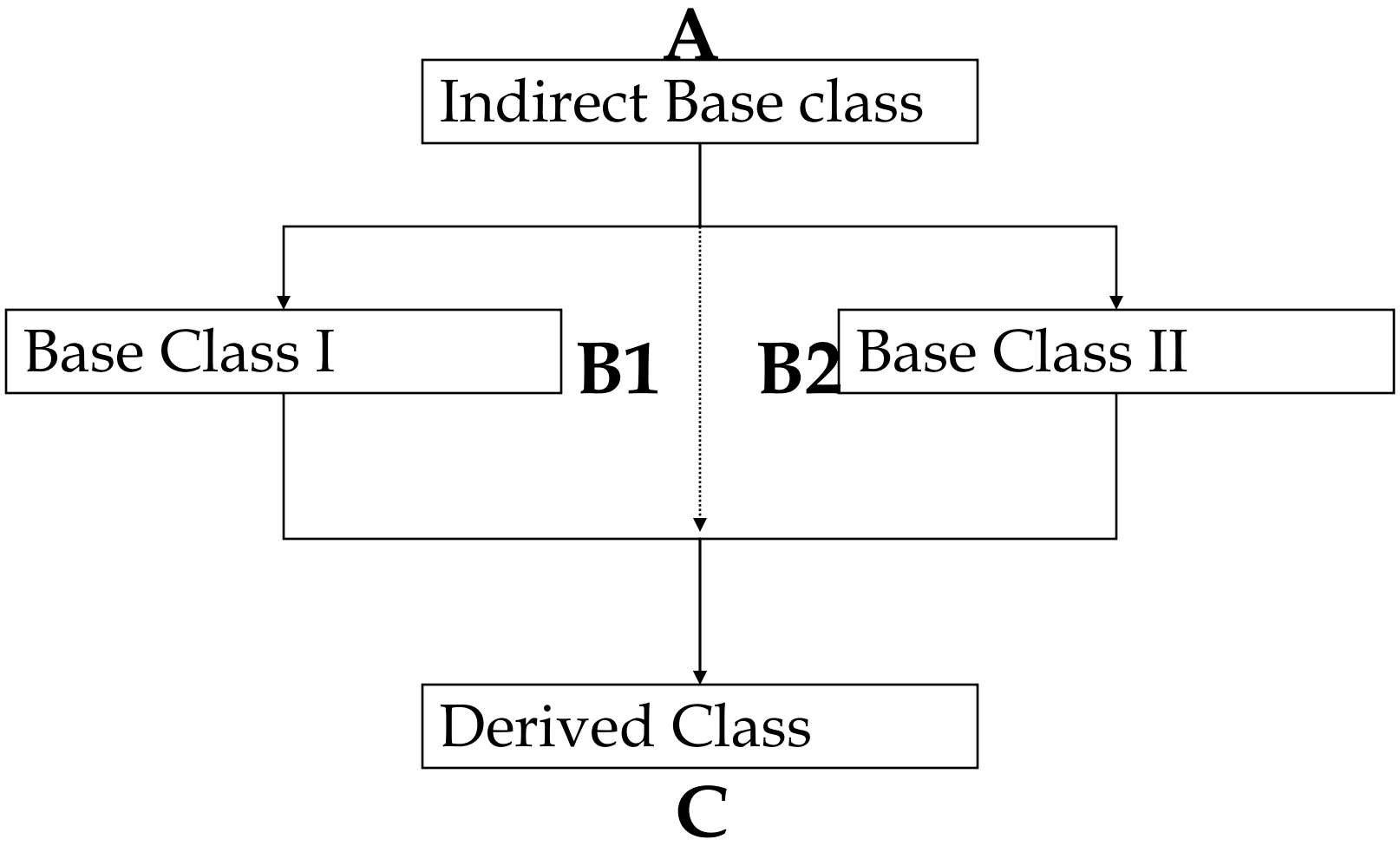
This is base's vfunc( )

This is derived1's vfunc( )

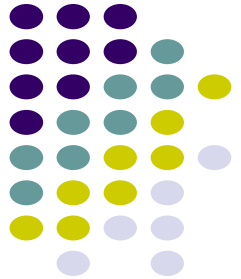
This is derived1's vfunc( )

Because inheritance is hierarchical in C++, it makes sense that virtual functions are also hierarchical. This means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used.

Hybrid inheritance



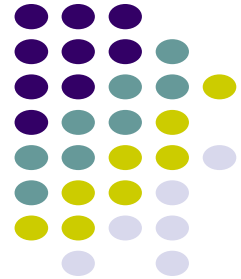
Diamond Problem



Multiple inheritance (continued)

The code that results in ambiguity is given below:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
class derived2 : public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
```

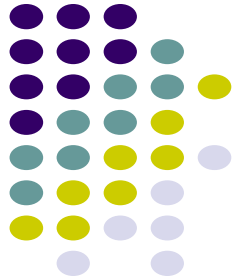


```
int main()
{
    derived3 ob;
    ob.i = 10; // this is ambiguous, which i???
    ob.j = 20;
    ob.k = 30;
    // i ambiguous here, too
    ob.sum = ob.i + ob.j + ob.k;
    // also ambiguous, which i?
    cout << ob.i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}
```

Compiler generated error :

```
$ c++ multi_inherit.c++
multi_inherit.c++:  In function âint main()â:
multi_inherit.c++:26:  error: request for member âiâ is ambiguous
multi_inherit.c++:5:  error: candidates are: int base::i
multi_inherit.c++:5:  error:                 int base::i
multi_inherit.c++:30:  error: request for member âiâ is ambiguous
multi_inherit.c++:5:  error: candidates are: int base::i
multi_inherit.c++:5:  error:                 int base::i
multi_inherit.c++:32:  error: request for member âiâ is ambiguous
multi_inherit.c++:5:  error: candidates are: int base::i
multi_inherit.c++:5:  error:                 int base::i
```

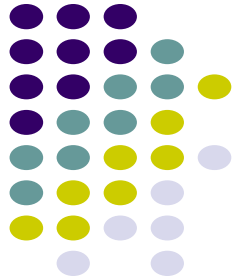
Multiple inheritance (continued)



There are two solutions for the ambiguity discussed in the previous slide. The first solution is:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
class derived2 : public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
```

Multiple inheritance (continued)



```
int main()
{
    derived3 ob;
    ob.derived1::i = 10; // scope resolved, use derived1's i
    ob.j = 20;
    ob.k = 30;
    // scope resolved
    ob.sum = ob.derived1::i + ob.j + ob.k;
    // also resolved here
    cout << ob.derived1::i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}
```

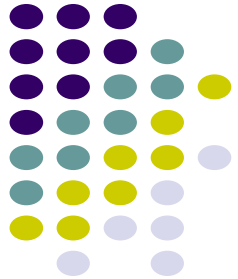
Output :

```
$ c++ multi_inherit_resolved.c++ ;./a.out
10 20 30 60
```

Explanation of code:

In this example code, *i* is the member of the Base class; two copies of this member *i* exists in the class derived3; One, derived from derived1 and other, from derived2; we can resolve the scope by either specifying *derived1::I* or *derived2::i*

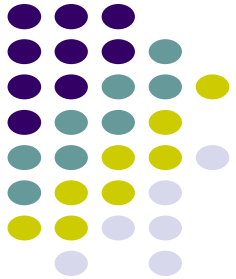
Multiple inheritance (continued)



Second solution using Virtual Base class is:

```
#include<iostream>
using namespace std;
class stdinfo {
        protected:
                int rollno;
                char name[10];
        public:
                void get_info();
};
void stdinfo::get_info() {
        cout<<"Enter rollno and name... ";
        cin>>rollno>>name;
}
class langmarks:virtual public stdinfo {
        protected:
                float s1,s2;
        public:
                void get_marks();
                float print_total();
};
void langmarks::get_marks() {
        cout<<"Enter marks in subjects s1 and s2... ";
        cin>>s1>>s2;
}
float langmarks::print_total() {
        return s1+s2;
}
```

Multiple inheritance (continued)

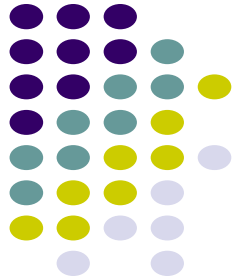


Second solution using Virtual Base class is: (continued)

```
class sports_marks:virtual public stdinfo {
        protected:
                float sp1;
        public:
                void get_sp_marks();
                float put_sp_marks();
};
void sports_marks::get_sp_marks() {
        cout<<"Enter sports marks... ";
        cin>>sp1;
}
float sports_marks::put_sp_marks() {
        return sp1;
}
```

```
class totmarks:public langmarks,public sports_marks {
        public:
                float compute_totmarks()                {
                        return print_total()+put_sp_marks();
                }
};
int main() {
        totmarks tm;
        tm.get_info(); //ambiguity avoided by declaring virtual base
        tm.get_marks();
        tm.get_sp_marks();
        cout<<"The total marks is "<<tm.compute_totmarks()<<endl;
        return 0;
}
```

Multiple inheritance (continued)



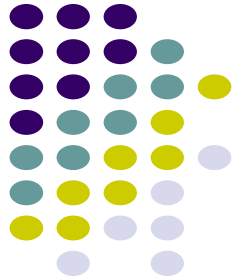
Second solution using Virtual Base class is: (continued)

Output :

```
$ c++ virtual_base_class.cpp ;./a.out
Enter rollno and name... 844
Mohankumar
Enter marks in subjects s1 and s2... 85
94
Enter sports marks... 79
The total marks is 258
```

Explanation of code:

In this example code, the function *get_info()* is member of the base class, stdinfo; the classes langmarks and sportmarks inherit this function from stdinfo; to avoid the class totmarks from inheriting both from langmarks and sportmarks, the stdinfo class is defined as virtual base class. Thus, ambiguity is resolved.



Virtual Base Class

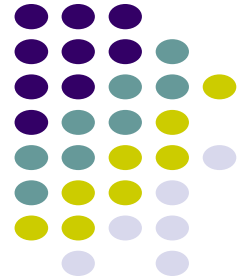
Properties of the virtual base class are made virtual for inheritance in the subsequent derived classes.

class B1:public virtual A

class B2:public virtual A

class C:public B1, public B2

- only one copy of the properties of class A will be inherited by class C via class B1 and class B2**.**



Polymorphism

Virtual Functions

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

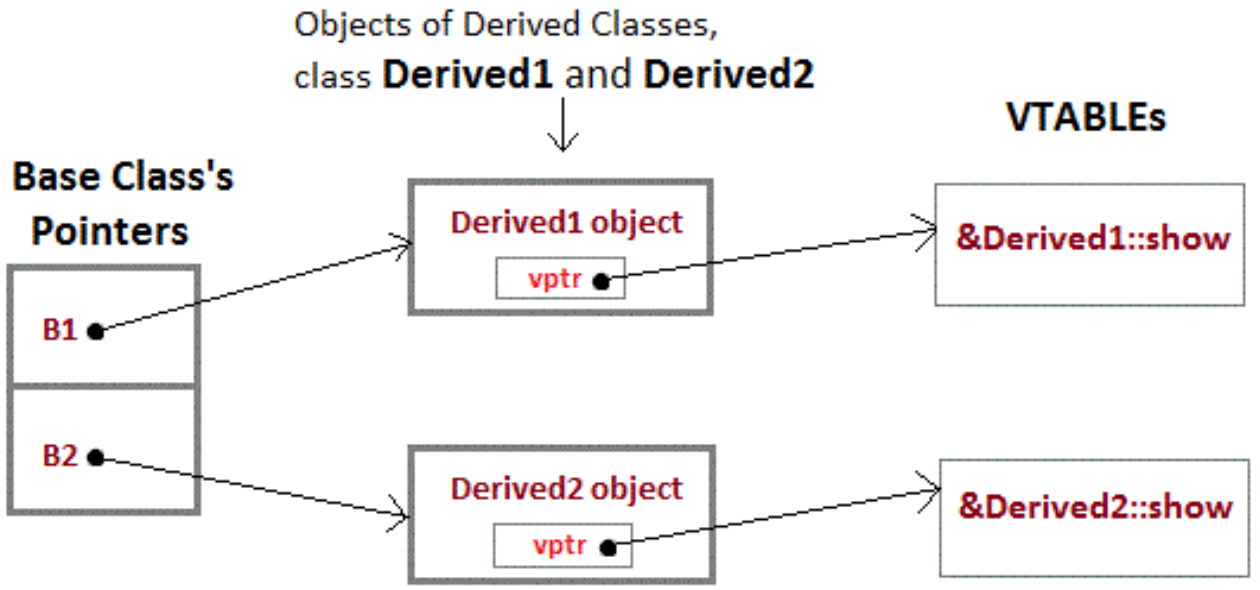
Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

example: take earlier example add virtual keyword in base class

Polymorphism

Virtual Functions



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

What is vTable?

The vTable, or Virtual Table, is a table of function pointers that is created by the compiler to support dynamic polymorphism. Whenever a class contains a virtual function, the compiler creates a Vtable for that class. Each object of the class is then provided with a hidden pointer to this table, known as Vptr.

The Vtable has one entry for each virtual function accessible by the class. These entries are pointers to the **most derived function** that the current object should call.

What is vPtr (Virtual Pointer)?

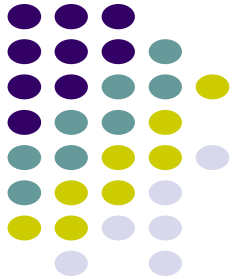
The virtual pointer or _vptr is a hidden pointer that is added by the compiler as a member of the class to point to the VTable of that class. For every object of a class containing virtual functions, a vptr is added to point to the vTable of that class. It's important to note that vptr is created only if a class has or inherits a virtual function.

Understanding vTable and vPtr Using an Example

Here is a simple example with a base class **Base** and classes **Derived1** derived from Base and **Derived2** from class **Derived1**.

```cpp
// C++ program to show the working of vtable and vptr
#include <iostream>
using namespace std;

// base class
class Base {
public:
    virtual void function1()
    {
        cout << "Base function1()" << endl;
    }
    virtual void function2()
    {
        cout << "Base function2()" << endl;
    }
    virtual void function3()
    {
        cout << "Base function3()" << endl;
    }
};

// class derived from Base
class Derived1 : public Base {
public:
    // overriding function1()
    void function1()
    {
        cout << "Derived1 function1()" << endl;
    }
    // not overriding function2() and function3()
};

// class derived from Derived1
class Derived2 : public Derived1 {
public:
    // again overriding function2()
    void function2()
    {
        cout << "Derived2 function2()" << endl;
    }
    // not overriding function1() and function3()
};
```

```cpp
// driver code
int main()
{
    // defining base class pointers
    Base* ptr1 = new Base();
    Base* ptr2 = new Derived1();
    Base* ptr3 = new Derived2();

    // calling all functions
    ptr1->function1();
    ptr1->function2();
    ptr1->function3();
    ptr2->function1();
    ptr2->function2();
    ptr2->function3();
    ptr3->function1();
    ptr3->function2();
    ptr3->function3();

    // deleting objects
    delete ptr1;
    delete ptr2;
    delete ptr3;

    return 0;
}
```



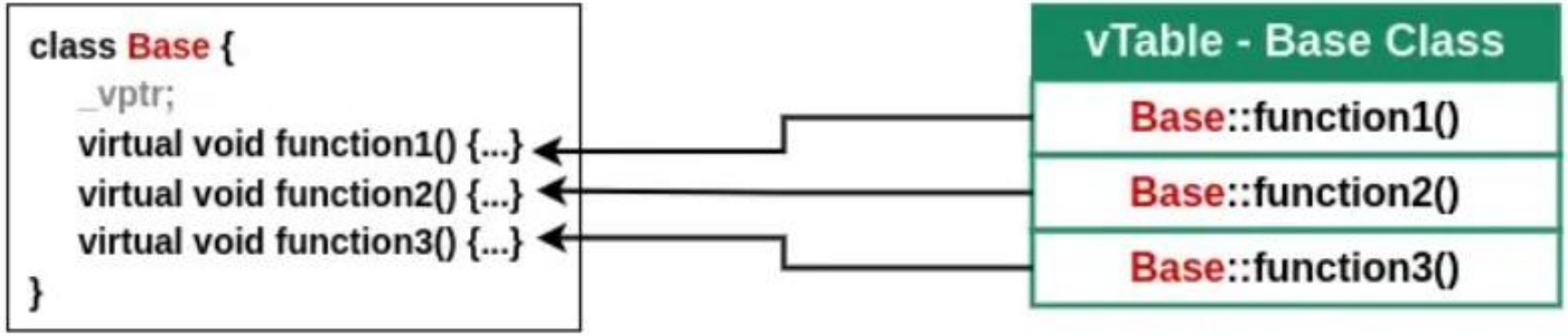
Output:

```
PS D:\aclDigital\CppProgramsACLPVT\ACLDPVT\Day04\Is-a_Has-a> .\a.exe
Base function1()
Base function2()
Base function3()
Derived1 function1()
Base function2()
Base function3()
Derived1 function1()
Derived2 function2()
Base function3()
PS D:\aclDigital\CppProgramsACLPVT\ACLDPVT\Day04\Is-a Has-a>
```

Explanation

1. For Base Class

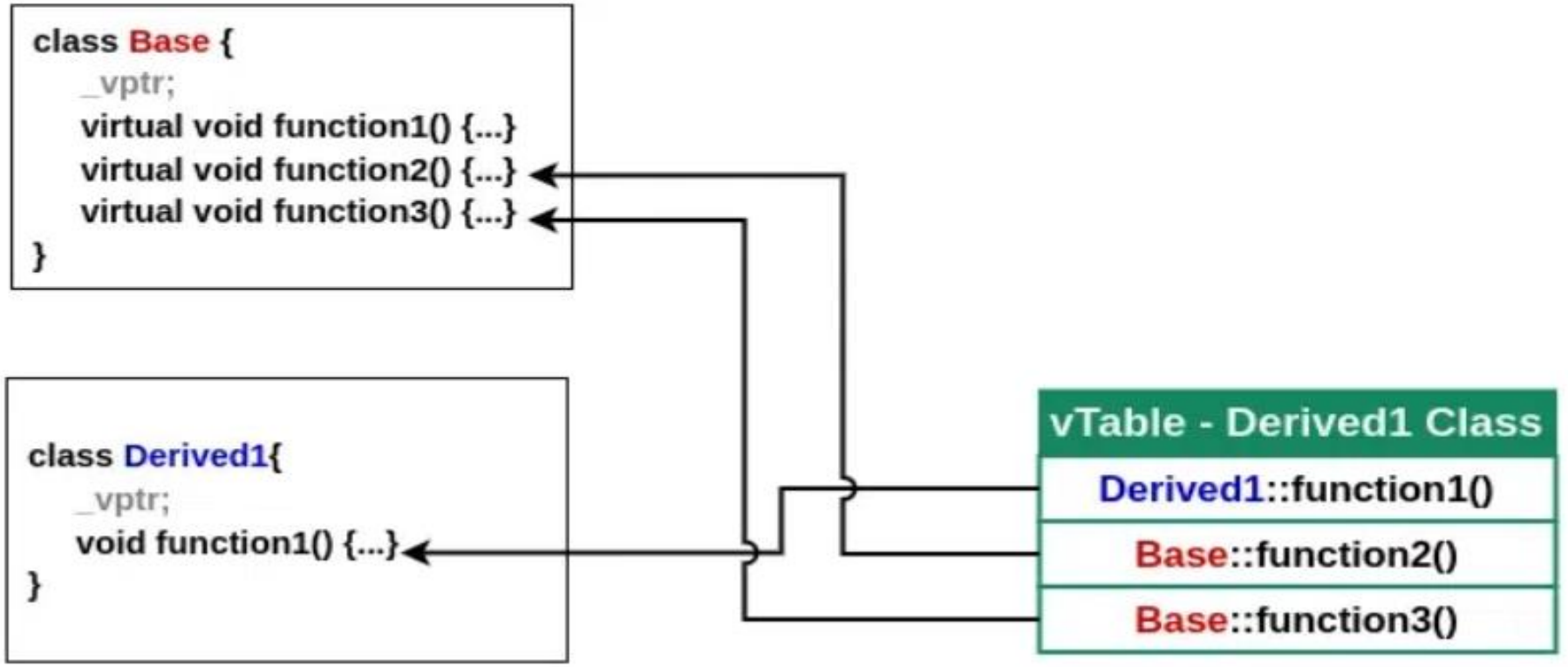
The **base class** have 3 virtual function **function1(), function2() and function3().** So the vTable of the base class would have 3 elements i.e. function pointer to base::function()



Base Class vTable

2. For Derived1 Class

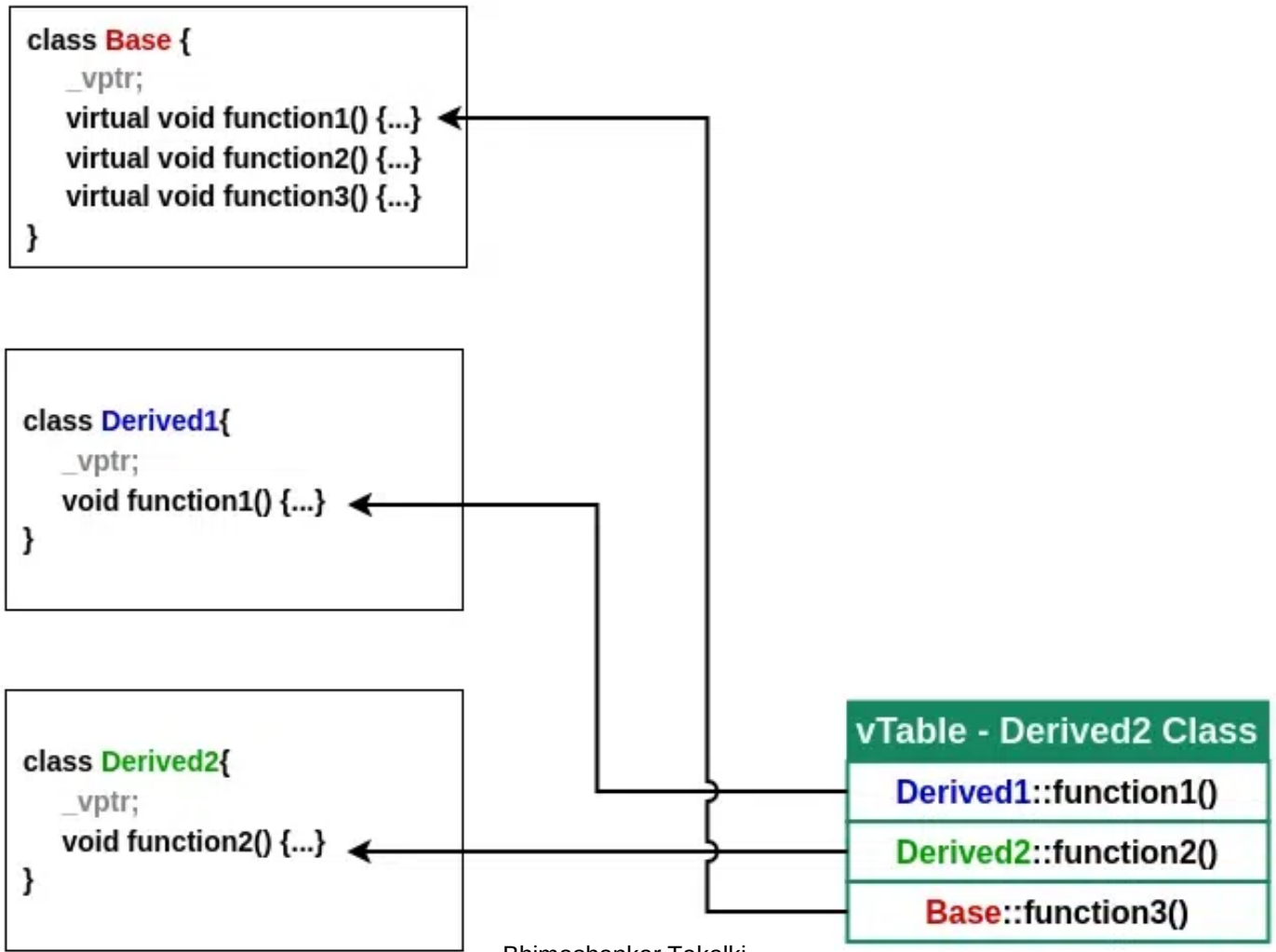
The **Derived1** class inherits 3 virtual functions from the **Base** class but overrides only **function1()** so all the other functions will be the same as the base class.



Derived1 Class vTable

3. For Derived2 Class

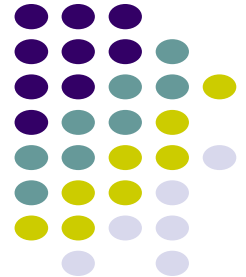
The **Derived2** class is inherited from **Derived1** class so it inherits the **function1() of Derived1** and **function2() and function3() of Base** class. In this class, we have overridden only function2(). So, the vTable for Derived2 class will look like this



Bhimashankar Takalki

Derived2 Class vTable

Advanced C++ | Virtual Constructor



Can we make a class constructor virtual in C++ to create polymorphic objects? No.

C++ being a statically typed (the purpose of RTTI is different) language, it is meaningless to the C++ compiler to create an object polymorphically.

The compiler must be aware of the class type to create the object. In other words, what type of object to be created is a compile-time decision from the C++ compiler perspective.

If we make a constructor virtual, the compiler flags an error. In fact, except inline, no other keyword is allowed in the declaration of the constructor.

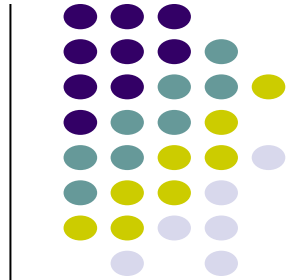
Implementation of Virtual Function

Real-Life Example to Understand the Implementation of Virtual Function
Consider employee management software for an organization.
Let the code has a simple base class Employee, the class contains virtual functions like raiseSalary(), transfer(), promote(), etc. Different types of employees like Managers, Engineers, etc., may have their own implementations of the virtual functions present in base class Employee.

In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, but we don’t need to worry about them because if raiseSalary() is present for a specific employee type, only that function would be called.

```
// C++ program to demonstrate how a virtual
function
// is used in a real life scenario

class Employee {
public:
        virtual void raiseSalary()
        {
                // common raise salary code
        }

        virtual void promote()
        {
                // common promote code
        }
};
```

```
class Manager : public Employee {
        virtual void raiseSalary()
        {
                // Manager specific raise salary code, may
contain
                // increment of manager specific
incentives
        }

        virtual void promote()
        {
                // Manager specific promote
        }
};
```

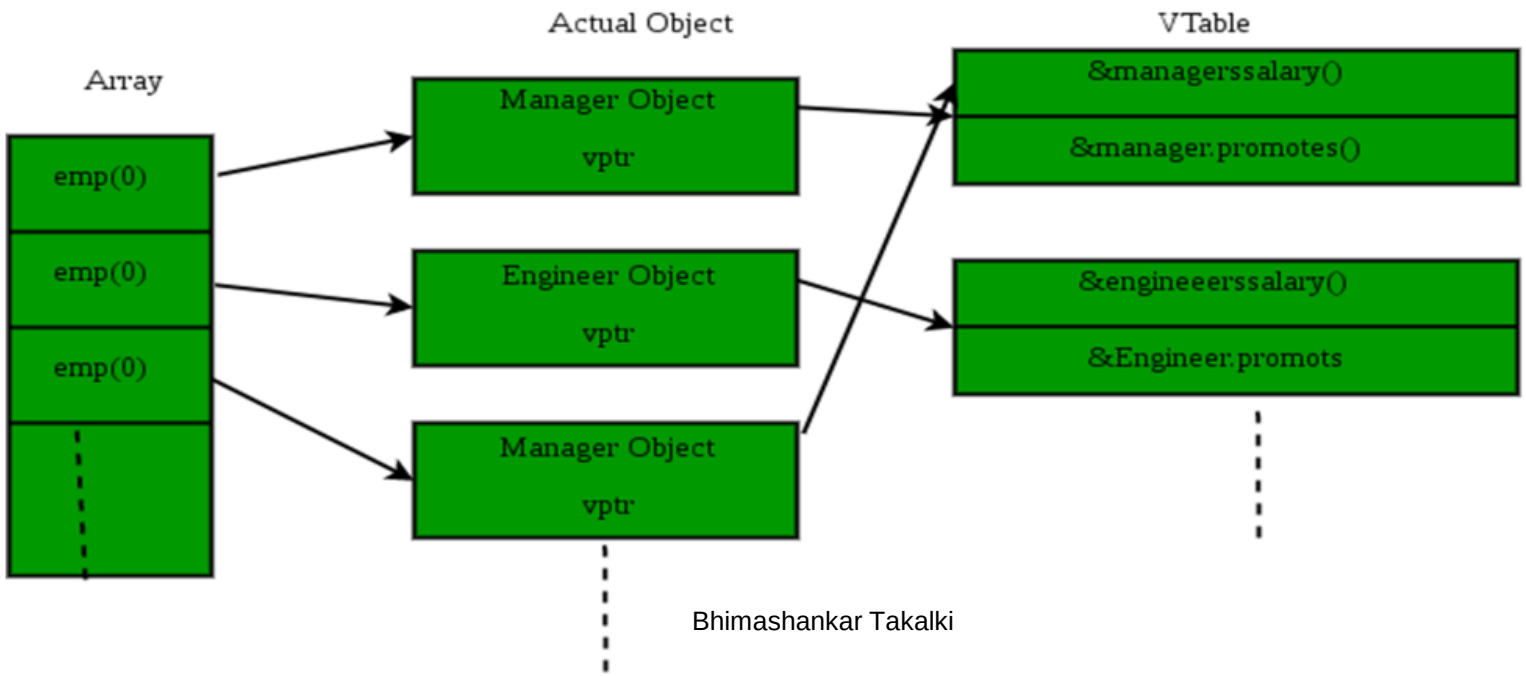


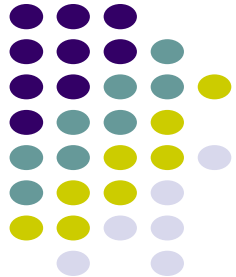
```
// Similarly, there may be other types of employees

// We need a very simple function
// to increment the salary of all employees
// Note that emp[] is an array of pointers
// and actual pointed objects can
// be any type of employees.
// This function should ideally
// be in a class like Organization,
// we have made it global to keep things simple
void globalRaiseSalary(Employee* emp[], int n)
{
        for (int i = 0; i < n; i++) {
                // Polymorphic Call: Calls raiseSalary()
                // according to the actual object, not
                // according to the type of pointer
                emp[i]->raiseSalary();
        }
}
```

Like the ‘globalRaiseSalary()‘ function, there can be many other operations that can be performed on a list of employees without even knowing the type of the object instance.
Virtual functions are so useful that later languages like Java keep all methods virtual by default.

How does the compiler perform runtime resolution?
The compiler maintains two things to serve this purpose:

vtable: A table of function pointers, maintained per class.
vptr: A pointer to vtable, maintained per object instance (see this for an example).





The compiler perform runtime resolution

The compiler adds additional code at two places to maintain and use vptr.

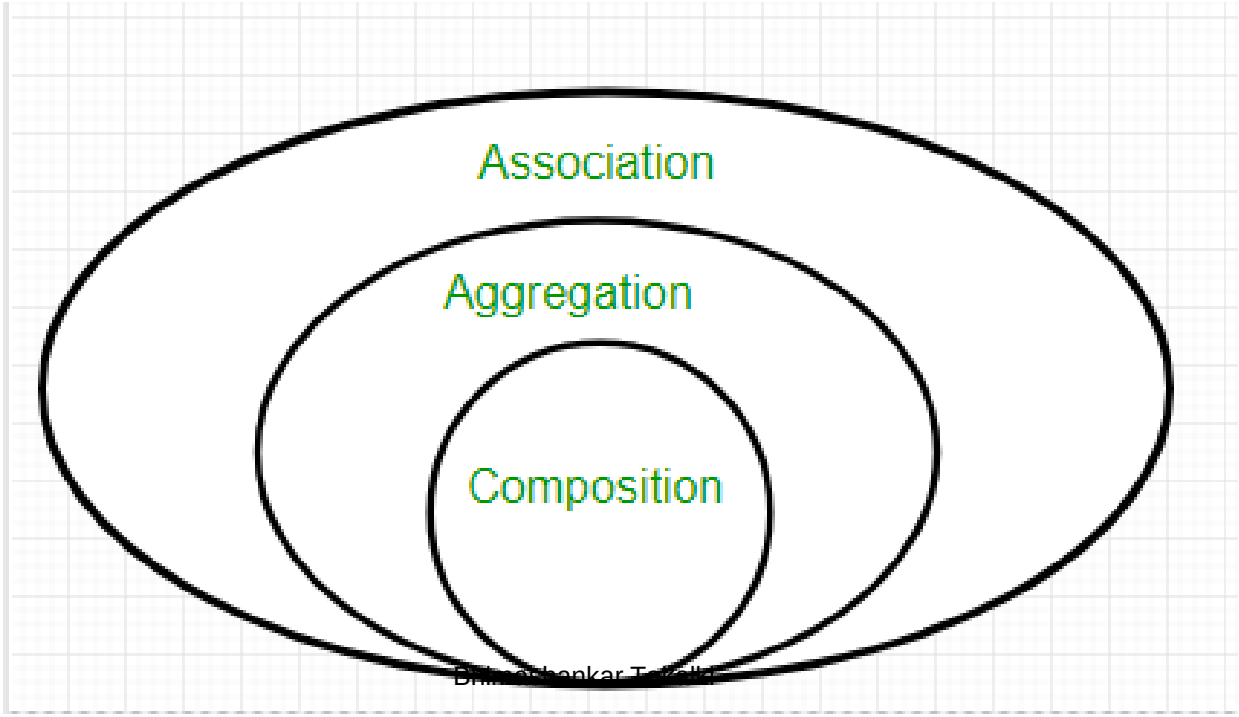
1. Code in every constructor. This code sets the vptr of the object being created. This code sets vptr to point to the vtable of the class.

2. Code with polymorphic function call (e.g. bp->show() in above code). Wherever a polymorphic call is made, the compiler inserts code to first look for vptr using a base class pointer or reference (In the above example, since the pointed or referred object is of a derived type, vptr of a derived class is accessed). Once vptr is fetched, vtable of derived class can be accessed. Using vtable, the address of the derived class function show() is accessed and called.

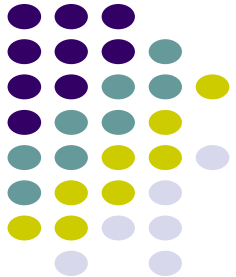
Is this a standard way for implementation of run-time polymorphism in C++?
The C++ standards do not mandate exactly how runtime polymorphism must be implemented, but compilers generally use minor variations on the same basic model.

Association, Composition and Aggregation in C++

Association is a relation between two separate classes which is established through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an Object communicates to another object to use functionality and services provided by that object. Composition and Aggregation are the two forms of association.



Association in C++



Association in C++ is a relationship between two classes where one class uses the functionalities provided by the other class. In other words, an association represents the connection or link between two classes. In an association, one class instance is connected to one or more instances of another class.

```
class Bank {
public:
    void transferMoney(Account* fromAccount, Account* toAccount, double amount) {
        // Code to transfer money from one account to another
    }
};

class Account {
public:
    Account(int id, double balance) : id(id), balance(balance) {}
    int getId() { return id; }
    double getBalance() { return balance; }
private:
    int id;
    double balance;
};

int main() {
    Account* account1 = new Account(123, 1000.00)
    Account* account2 = new Account(456, 500.00);
    Bank bank;
    bank.transferMoney(account1, account2, 250.00);
    return 0;
}
```

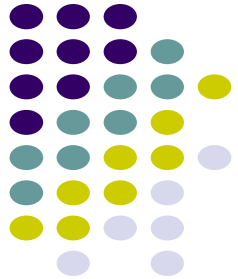
Association in C++

In this example, we have two classes: Bank and Account. Bank class has a transferMoney() method that takes two Account objects and a double value representing the amount of money to be transferred from one account to another. Bank class is associated with the Account class, meaning that it has a connection or a link to the Account class.

In the main() function, we create two Account objects with different IDs and balances, and then we create an instance of the Bank class. We then call the transferMoney() method of the Bank class and pass it the two Account objects and the amount of money to be transferred.

As we can see, the Bank class has a direct association with the Account class because it uses objects of the Account class as parameters in its method.

Composition in C++

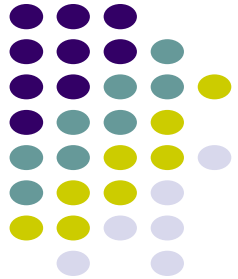


Composition:

Composition is a relationship between two classes in which one class, known as the composite class, contains an object of another class, known as the component class, as a member variable. The composite class owns the component class, and the component class cannot exist independently of the composite class. In other words, the lifetime of the component class is controlled by the composite class.

```
class Engine {
public:
    void start() {
        // Code to start the engine
    }
};

class Car {
public:
    Car() : engine(new Engine()) {}
    void startCar() {
        engine->start();
    }
private:
    Engine* engine;
};

int main() {
    Car car;
    car.startCar();
    return 0;
}
```

Composition in C++

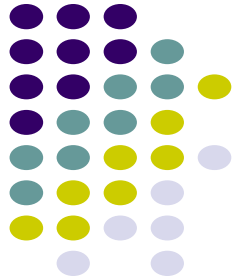


In this example, we have two classes: Engine and Car. The Car class is composed of an Engine object. The Engine object is an essential part of the Car object, and if the Car object is destroyed, the Engine object will also be destroyed.

In the Car class constructor, we create a new instance of the Engine class and store it in a pointer variable named engine. We then provide a method named startCar() in the Car class, which calls the start() method of the Engine object stored in the engine pointer.

In the main() function, we create a new instance of the Car class and call the startCar() method to start the car. As we can see, the Car class is composed of an Engine object, and the startCar() method of the Car class uses the Engine object to start the car.

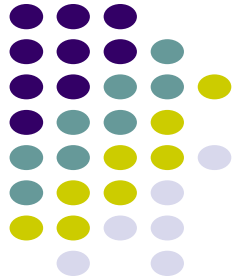
Aggregation in C++



Aggregation is a relationship between two classes in which one class, known as the aggregate class, contains a pointer or reference to an object of another class, known as the component class.

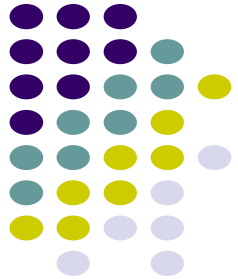
The component class can exist independently of the aggregate class, and it can be shared by multiple aggregate classes. In other words, the lifetime of the component class is not controlled by the aggregate class.

Aggregation in C++



```
class Person {
public:
    Person(std::string name) : name(name), address(nullptr) {}
    void setAddress(Address* address) {
        this->address = address;
    }
private:
    std::string name;
    Address* address;
};

class Address {
public:
    Address(std::string street, std::string city, std::string state, std::string zip)
        : street(street), city(city), state(state), zip(zip) {}
private:
    std::string street;
    std::string city;
    std::string state;
    std::string zip;
};
```

Aggregation in C++



```
int main() {
    Address* address = new Address("123 Main St.", "Anytown", "CA", "12345");
    Person person("John Doe");
    person.setAddress(address);
    return 0;
}
```

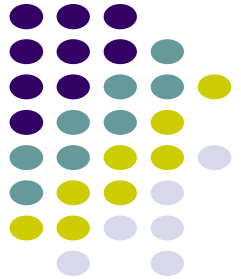
Aggregation in C++

In this example, we have two classes: Person and Address. The Person class has a member variable named address, which is a pointer to an Address object. The Address object is not an essential part of the Person object, and if the Person object is destroyed, the Address object will not be destroyed.

In the main() function, we create a new Address object and store it in a pointer variable named address. We then create a new instance of the Person class and pass it the name “John Doe”. We then call the setAddress() method of the Person class and pass it the address pointer. This sets the address of the Person object to the address object we created earlier.

As we can see, the Person class has an aggregation relationship with the Address class because the Person object contains a pointer to the Address object, but the Address object is not an essential part of the Person object.

Inheritance versus composition: Is-a and has-a relationships



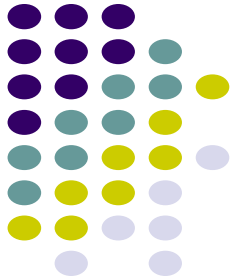
Is-a relationship is inheritance and used to extend the capability of base.

- Example of is-a relationship:
  - An engineer ***is an*** employee.
  - A clerk ***is an*** employee.
  - A car ***is a*** four wheeler.

Has-a relationship represents composition. This relationship can also be indicated by the term containership.

- Example of has-a relationship:
  - A book *has-a* price.
  - A book *has-an* ISBN No.
  - A pen *class has* a ball class.

Inheritance versus composition: Is-a and has-a relationships (continued)

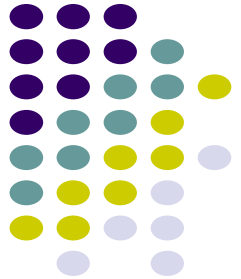


Has-a relationship can be established using simple composition, as depicted in the following example code snippet:

- The “car has-a Engine" relationship can be expressed using *simple composition*:

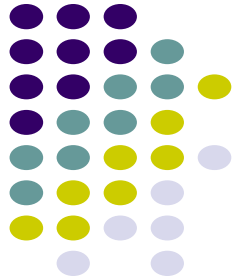
```
class Engine   {
        public:
          Engine (int numCylinders);
            void start();                    // Starts this Engine
        };
class Car   {
        public:
          Car() : e_(8) { }                  // Initializes this Car with 8 cylinders
          void start() { e_.start(); }       // Start this Car by starting its Engine
        private:
          Engine e_;                         // Car has-a Engine
        };
```

Inheritance versus composition: Is-a and has-a relationships (continued)



- The Car has-an Engine can also be expressed by using private inheritance, as illustrated in the following code snippet:

```
class Car : private Engine
{                                   // Car has-a Engine
        public:
          Car() : Engine(8) { }     // Initializes this Car with 8 cylinders
          using Engine::start;      // Start this Car by starting its Engine
                                    // Expose the hidden member–Publicizer Technique.
};
```

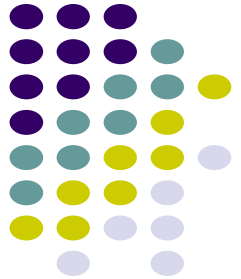


Pure Virtual Functions

- When a virtual function is not redefined by the derived class, the version defined in the base class will be used. However, in many situations, there cannot be any meaningful definition of a virtual function within a base class.

- For example, a base class may not be able to define an object sufficiently to allow a base class virtual function to be created.

- Further, in some situations, you will want to ensure that all derived classes compulsorily override a virtual function.



Pure Virtual Functions

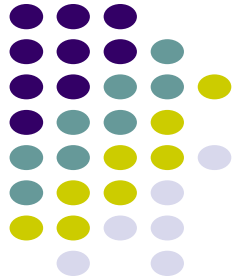
To handle these two situations, C++ supports the **Pure Virtual Function.** A pure virtual function is a virtual function that has no definition in the base class.

To declare a pure virtual function, use this general form:

virtual type func_name (parameter_list) = 0;

When a virtual function is declared pure, any derived class **must** provide its own definition of the virtual function. If the derived class fails to override the pure virtual function, a compile-time error will ensue.

Pure Virtual Functions



The base class **number** contains an integer called **val,** the function setval( ) and the pure virtual function show( )

The derived class hextype, dectype, and octtype inherit number, and redefine show( ) so that it outputs the value of val in each number base (i.e., hexadecimal, decimal or octal).

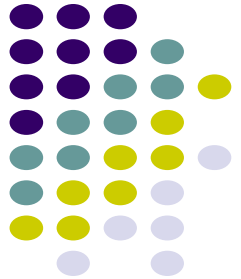
```
#include<iostream>
using namespace std;
class number
 {
   protected:
    int val;
   public:
    void setval( int i)
     { val = i;}
    virtual void show( ) = 0;
};
```

```
class hextype : number
 {
   public:
    void show( )
     { cout << hex << val << "\n"; } };

class dectype : number
 { public:
    void show( )
     { cout << val << "\n"; } };
```

```
class octtype : number
 { public:
    void show( )
     { cout << oct << val << "\n"; }
};
```

Pure Virtual Functions

```
int main( )
 {
      number *p;
      dectype d;
      hextype h;
      octtype o;
      p = &d;
      d.setval(20);
      p->show( ); // displays 20 – decimal
      p = &h;
      h.setval(20);
      p->show( ); // displays 14 – hexadecimal
      p = &o;
     o.setval(20);
     p->show( ); // displays 24 –octal
     return 0;
}
```



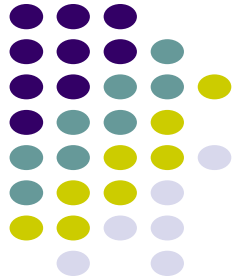
Abstract Classes

- A class that contains at least one pure virtual function is said to be abstract. An abstract class cannot be instantiated since it has one or more pure virtual functions.

- Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class.

- This allows abstract classes to support runtime polymorphism, which relies upon base class pointers or references to select the proper virtual function.

Abstract Classes - Using Virtual Functions

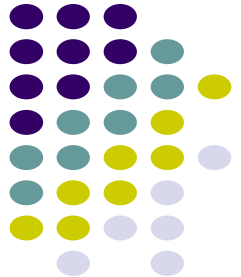


- One of the central aspects of Object-Oriented Programming is the principle of “one interface, multiple methods”.

- This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations.

- In concrete C++ terms, the base class can be used to define the nature of the interface to a general class.

Abstract Classes - Using Virtual Functions

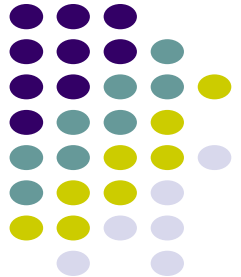


- Each derived class then implements the specific operations as they relate to the type of data used by the derived type.

- One of the most powerful and flexible ways to implement the “one interface, multiple methods” approach is to use abstract base classes, pure virtual functions, and base class references or pointers.

- Using these features, you can define a class hierarchy that moves from general to the specific (base to derived).

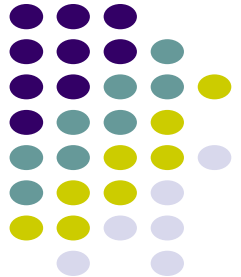
Abstract Classes - Using Virtual Functions



- define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function.

- Therefore, in the base class, you create and define everything you can that relates to the general class. The derived class fills in the specific details.

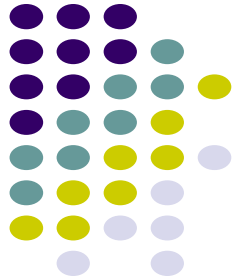
Abstract Classes - Using Virtual Functions



- define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function.

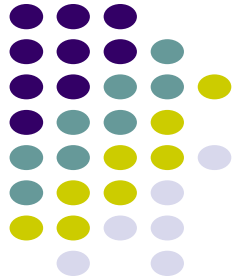
- Therefore, in the base class, you create and define everything you can that relates to the general class. The derived class fills in the specific details.

Example: Abstract class and Virtual Function in calculating Area of Square and Circle:



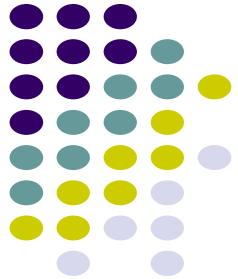
```
// C++ program to calculate the area of a square and a circle
// Abstract class
class Shape {
    protected:
        float dimension;
    public:
        void getDimension() {
        cin >> dimension;
        }
        // pure virtual Function
        virtual float calculateArea() = 0;
};
// Derived class
class Square : public Shape {
    public:
        float calculateArea() {
            return dimension * dimension;
        }
};
```

Example: Abstract class and Virtual Function in calculating Area of Square and Circle:



```
// Derived class
class Circle : public Shape {
    public:
        float calculateArea() {
            return 3.14 * dimension * dimension;
        }
};

int main()
{
    Square square;
    Circle circle;
    cout << "Enter the length of the square: ";
    square.getDimension();
    cout << "Area of square: " << square.calculateArea() << endl;
    cout << "\nEnter radius of the circle: ";
    circle.getDimension();
    cout << "Area of circle: " << circle.calculateArea() << endl;
    return 0;
}
```

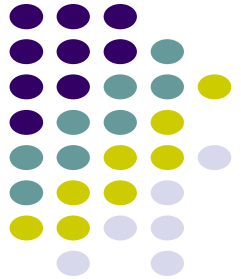
Example: Abstract class and Virtual Function in calculating Area of Square and Circle:



Output:

```
Enter length to calculate the area of a square: 4
Area of square: 16
Enter radius to calculate the area of a circle: 5
Area of circle: 78.5
```

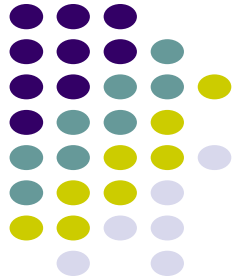
Virtual Destructor



What is a Virtual Destructor?

- A destructor is a special member function in C++ that is called when an object goes out of scope or is explicitly deleted.
- When dealing with inheritance, if a base class pointer points to a derived class object and the destructor is not virtual, the derived class destructor will not be called, leading to resource leaks.

Virtual Destructor



Why Use a Virtual Destructor?

- **Correct Resource Cleanup:** Ensures the derived class destructor is called, properly releasing resources.
- **Polymorphic Deletion:** Necessary for classes intended to be used polymorphically.

```
#include <iostream>

class Base {
public:
    Base() { std::cout << "Base Constructor\n"; }
    virtual ~Base() { std::cout << "Base Destructor\n"; }  // Virtual Destructor
};

class Derived : public Base {
public:
    Derived() { std::cout << "Derived Constructor\n"; }
    ~Derived() { std::cout << "Derived Destructor\n"; }
};

int main() {
    Base* obj = new Derived();  // Base pointer to Derived object
    delete obj;                 // Correctly calls both destructors
    return 0;
}
```

Output:

Base Constructor
Derived Constructor
Derived Destructor
Base Destructor

Key Points to Remember:

- Always declare destructors virtual in base classes intended for inheritance.
- Virtual destructors prevent memory leaks and ensure proper cleanup in derived classes.