



VIT[®]

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

BCSE102L –Structured and Object oriented Programming (Theory)

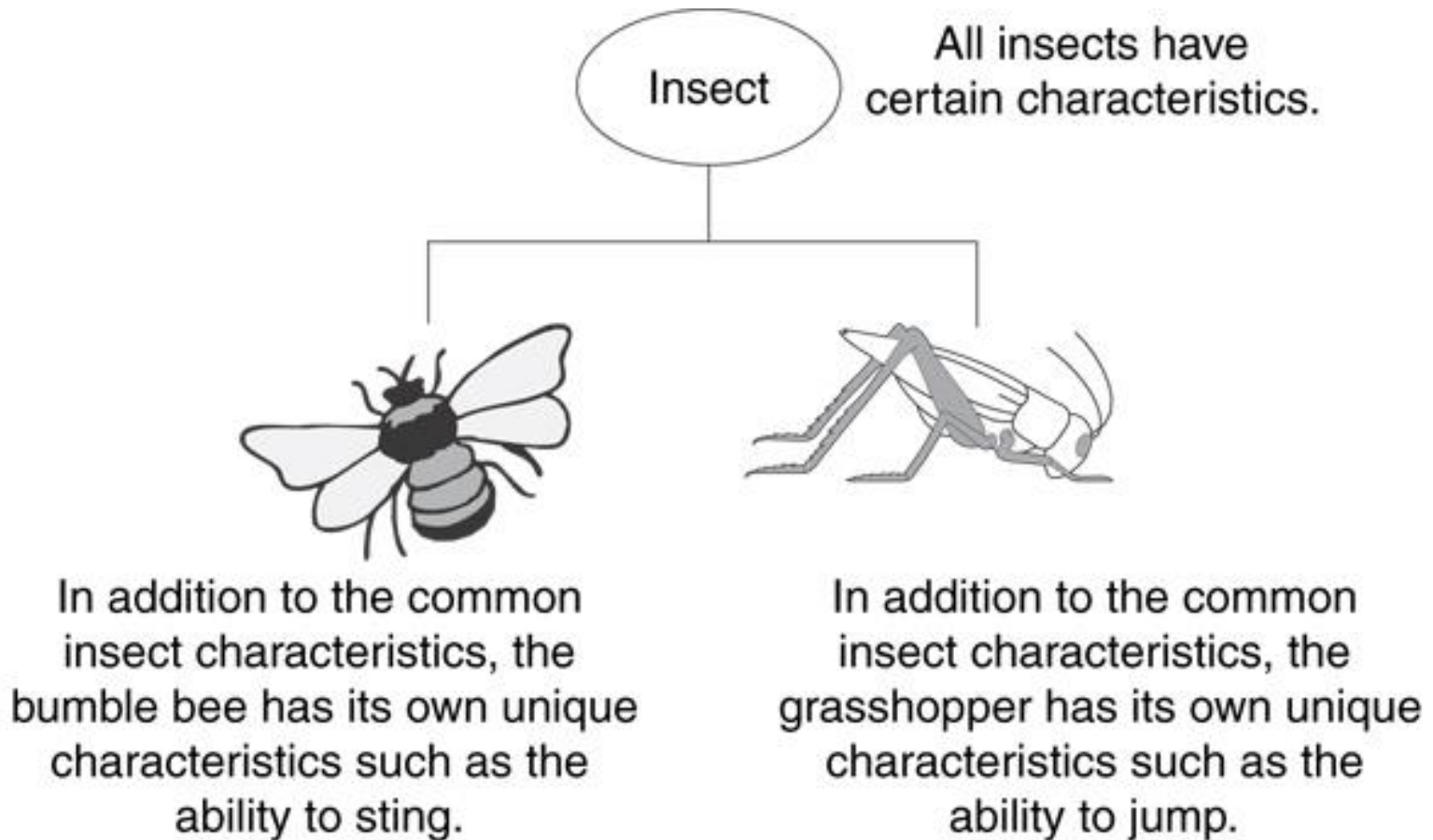
Module VI

Inheritance

What Is Inheritance?

- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class

Example: Insect Taxonomy



The "is a" Relationship

- Inheritance establishes an "is a" relationship between classes.
 - A poodle is a dog
 - A car is a vehicle
 - A flower is a plant
 - A football player is an athlete

Inheritance – Terminology and Notation in C++

- Base class (or parent) – inherited from
- Derived class (or child) – inherits from the base class
- Notation:

```
class Student // base class
{
    . . .
};
class UnderGrad : public student
{ // derived class
    . . .
};
```

Back to the 'is a' Relationship

- An object of a derived class 'is a(n)' object of the base class
- Example:
 - an UnderGrad is a Student
 - a Mammal is an Animal
- A derived object has **all** of the characteristics of the base class

What Does a Child Have?

An object of the derived class has:

- all members defined in child class
- all members declared in parent class

An object of the derived class can use:

- all `public` members defined in child class
- all `public` members defined in parent class

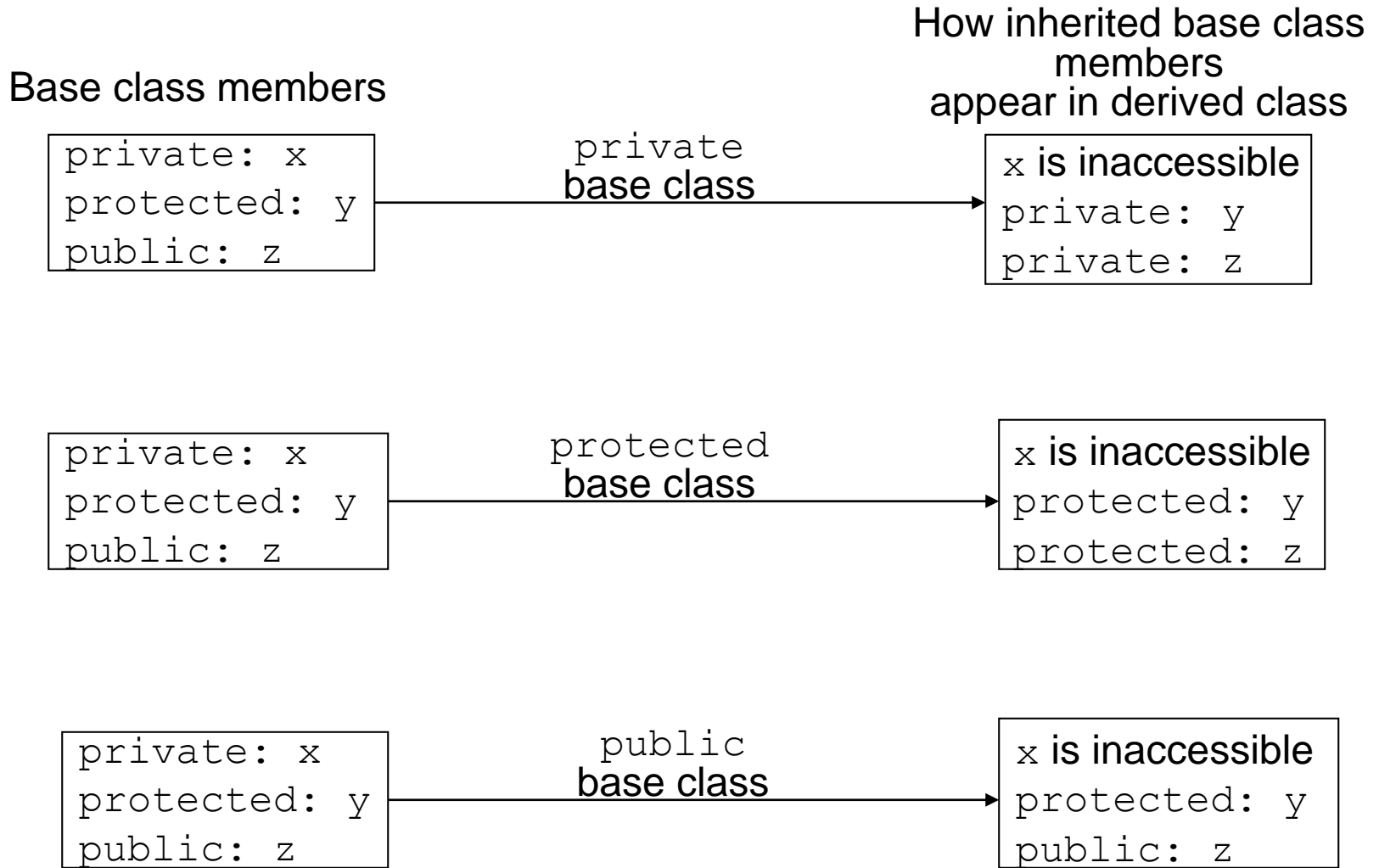
Protected Members and Class Access

- protected member access specification: like `private`, but accessible by objects of derived class
- Class access specification: determines how `private`, `protected`, and `public` members of base class are inherited by the derived class

Class Access Specifiers

- 1) `public` – object of derived class can be treated as object of base class (not vice-versa)
- 2) `protected` – more restrictive than `public`, but allows derived classes to know details of parents
- 3) `private` – prevents objects of derived class from being treated as objects of base class.

Inheritance vs. Access



Inheritance vs. Access

class Grade

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```

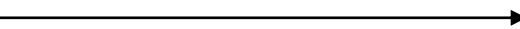
class Test : public Grade

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

When Test class inherits
from Grade class using
public class access, it
looks like this: 

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);  
void setScore(float);  
float getScore();  
char getLetter();
```

Inheritance vs. Access

class Grade

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```

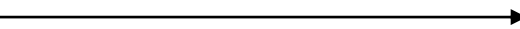
class Test : protected Grade

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

When Test class inherits
from Grade class using
protected class access, it
looks like this: 

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

protected members:

```
void setScore(float);  
float getScore();  
float getLetter();
```

Inheritance vs. Access

class Grade

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```


class Test : private Grade

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

When Test class inherits
from Grade class using
private class access, it
looks like this: 

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;  
void setScore(float);  
float getScore();  
float getLetter();
```

public members:

```
Test(int, int);
```

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Constructors and Destructors in Base and Derived Classes

Program 15-4

```
1 // This program demonstrates the order in which base and
2 // derived class constructors and destructors are called.
3 #include <iostream>
4 using namespace std;
5
6 //*****
7 // BaseClass declaration          *
8 //*****
9
```

Constructors and Destructors in Base and Derived Classes

Program 15-4 (continued)

```
10 class BaseClass
11 {
12 public:
13     BaseClass() // Constructor
14         { cout << "This is the BaseClass constructor.\n"; }
15
16     ~BaseClass() // Destructor
17         { cout << "This is the BaseClass destructor.\n"; }
18 };
19
20 //*****
21 // DerivedClass declaration      *
22 //*****
23
24 class DerivedClass: public BaseClass
25 {
26 public:
27     DerivedClass() // Constructor
28         { cout << "This is the DerivedClass constructor.\n"; }
29
30     ~DerivedClass() // Destructor
31         { cout << "This is the DerivedClass destructor.\n"; }
32 };
33
```


Constructors and Destructors in Base and Derived Classes

```
34 //*****
35 // main function *
36 //*****
37
38 int main()
39 {
40     cout << "We will now define a DerivedClass object.\n";
41
42     DerivedClass object;
43
44     cout << "The program is now going to end.\n";
45     return 0;
46 }
```

Program Output

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:

```
Square::Square(int side) :  
                        Rectangle(side,  
side)
```

- Can also be done with inline constructors
- Must be done if base class has no default constructor

Passing Arguments to Base Class Constructor

derived class constructor

base class constructor

```
Square::Square(int side) : Rectangle(side, side)
```

derived constructor parameter

base constructor parameters

Redefining Base Class Functions

- Redefining function: function in a derived class that has the *same name and parameter list* as a function in the base class
- Typically used to replace a function in base class with different actions in derived class

Redefining Base Class Functions

- Not the same as overloading – with overloading, parameter lists must be different
- Objects of base class use base class version of function; objects of derived class use derived class version of function

Base Class

```
class GradedActivity
{
protected:
    char letter;           // To hold the letter grade
    double score;         // To hold the numeric score
    void determineGrade(); // Determines the letter grade
public:
    // Default constructor
    GradedActivity()
        { letter = ' '; score = 0.0; }

    // Mutator function
    void setScore(double s)
        { score = s;
          determineGrade();}

    // Accessor functions
    double getScore() const
        { return score; }

    char getLetterGrade() const
        { return letter; }
};
```

Derived Class

```
1 #ifndef CURVEDACTIVITY_H
2 #define CURVEDACTIVITY_H
3 #include "GradedActivity.h"
4
5 class CurvedActivity : public GradedActivity
6 {
7 protected:
8     double rawScore;    // Unadjusted score
9     double percentage;  // Curve percentage
10 public:
11     // Default constructor
12     CurvedActivity() : GradedActivity()
13         { rawScore = 0.0; percentage = 0.0; }
14
15     // Mutator functions
16     void setScore(double s)           Redefined setScore function
17         { rawScore = s;
18           GradedActivity::setScore(rawScore * percentage); }
19
20     void setPercentage(double c)
21         { percentage = c; }
22
23     // Accessor functions
24     double getPercentage() const
25         { return percentage; }
26
27     double getRawScore() const
28         { return rawScore; }
29 };
30 #endif
```

Driver Program

```
13 // Define a CurvedActivity object.
14 CurvedActivity exam;
15
16 // Get the unadjusted score.
17 cout << "Enter the student's raw numeric score: ";
18 cin >> numericScore;
19
20 // Get the curve percentage.
21 cout << "Enter the curve percentage for this student: ";
22 cin >> percentage;
23
24 // Send the values to the exam object.
25 exam.setPercentage(percentage);
26 exam.setScore(numericScore);
27
28 // Display the grade data.
29 cout << fixed << setprecision(2);
30 cout << "The raw score is "
31     << exam.getRawScore() << endl;
32 cout << "The curved score is "
33     << exam.getScore() << endl;
34 cout << "The curved grade is "
35     << exam.getLetterGrade() << endl;
```

Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87 [Enter]
Enter the curve percentage for this student: 1.06 [Enter]
The raw score is 87.00
The curved score is 92.22
The curved grade is A
```


Problem with Redefining

- Consider this situation:
 - Class `BaseClass` defines functions `x()` and `y()`.
`x()` calls `y()`.
 - Class `DerivedClass` inherits from `BaseClass` and redefines function `y()`.
 - An object `D` of class `DerivedClass` is created and function `x()` is called.
 - When `x()` is called, which `y()` is used, the one defined in `BaseClass` or the the redefined one in `DerivedClass`?

Problem with Redefining

BaseClass

```
void X();  
void Y();
```

DerivedClass

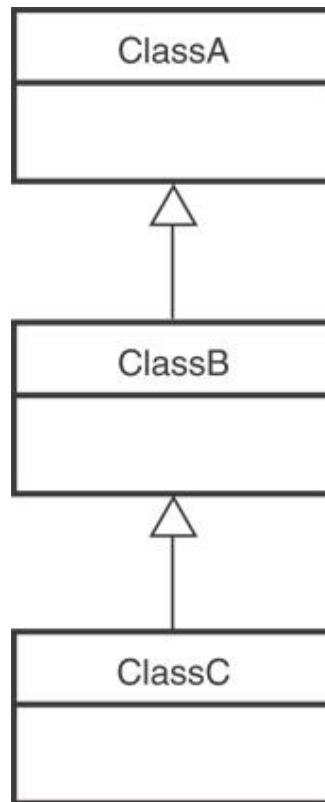
```
void Y();
```

```
DerivedClass D;  
D.X();
```

Object D invokes function X ()
In BaseClass. Function X ()
invokes function Y () in BaseClass, not
function Y () in DerivedClass,
because function calls are bound at
compile time. This is static binding.

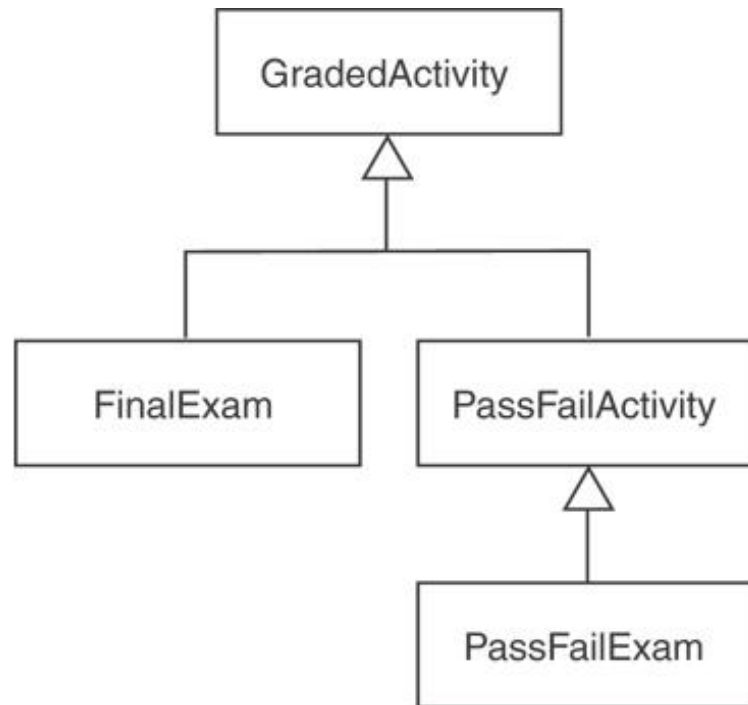
Class Hierarchies

- A base class can be derived from another base class.



Class Hierarchies

- Consider the GradedActivity, FinalExam, PassFailActivity, PassFailExam hierarchy in Chapter 15.



Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class
- Function defined with key word `virtual`:

```
virtual void Y() {...}
```
- Supports dynamic binding: functions bound at run time to function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Polymorphism and Virtual Member Functions

```
29 void displayGrade(const GradedActivity &activity)
30 {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33         << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35         << activity.getLetterGrade() << endl;
36 }
```

Because the parameter in the `displayGrade` function is a `GradedActivity` reference variable, it can reference any object that is derived from `GradedActivity`. That means we can pass a `GradedActivity` object, a `FinalExam` object, a `PassFailExam` object, or any other object that is derived from `GradedActivity`.

A problem occurs in Program 15-10 however...

Program 15-10

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailActivity.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity &);
8
9  int main()
10 {
11     // Create a PassFailActivity object. Minimum passing
12     // score is 70.
13     PassFailActivity test(70);
14
15     // Set the score to 72.
16     test.setScore(72);
17
18     // Display the object's grade data. The letter grade
19     // should be 'P'. What will be displayed?
20     displayGrade(test);
21     return 0;
22 }
```

```

23
24 //*****
25 // The displayGrade function displays a GradedActivity object's *
26 // numeric score and letter grade.                               *
27 //*****
28
29 void displayGrade(const GradedActivity &activity)
30 {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33         << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35         << activity.getLetterGrade() << endl;
36 }

```

Program Output

```

The activity's numeric score is 72.0
The activity's letter grade is C

```

As you can see from the example output, the `getLetterGrade` member function returned 'C' instead of 'P'. This is because the `GradedActivity` class's `getLetterGrade` function was executed instead of the `PassFailActivity` class's version of the function.

Static Binding

- Program 15-10 displays 'C' instead of 'P' because the call to the `getLetterGrade` function is statically bound (at compile time) with the `GradedActivity` class's version of the function.

We can remedy this by making the function *virtual*.

Virtual Functions

- A virtual function is dynamically bound to calls at runtime.

At runtime, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.

Virtual Functions

- To make a function virtual, place the virtual key word before the return type in the base class's declaration:

```
virtual char getLetterGrade() const;
```

- The compiler will not bind the function to calls. Instead, the program will bind them at runtime.

Updated Version of GradedActivity

```
6 class GradedActivity
7 {
8 protected:
9     double score; // To hold the numeric score
10 public:
11     // Default constructor
12     GradedActivity()
13         { score = 0.0; }
14
15     // Constructor
16     GradedActivity(double s)
17         { score = s; }
18
19     // Mutator function
20     void setScore(double s)
21         { score = s; }
22
23     // Accessor functions
24     double getScore() const
25         { return score; }
26
27     virtual char getLetterGrade() const;
28 };
```

The function
is now virtual.

The function also becomes
virtual in all derived classes
automatically!

Polymorphism

If we recompile our program with the updated versions of the classes, we will get the right output, shown here:
(See Program 15-11 in the book.)

Program Output

```
The activity's numeric score is 72.0  
The activity's letter grade is P
```

This type of behavior is known as polymorphism. The term *polymorphism* means the ability to take many forms.

Program 15-12 demonstrates polymorphism by passing objects of the `GradedActivity` and `PassFailExam` classes to the `displayGrade` function.

Program 15-12

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailExam.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity &);
8
9  int main()
10 {
11     // Create a GradedActivity object. The score is 88.
12     GradedActivity test1(88.0);
13
14     // Create a PassFailExam object. There are 100 questions,
15     // the student missed 25 of them, and the minimum passing
16     // score is 70.
17     PassFailExam test2(100, 25, 70.0);
18
19     // Display the grade data for both objects.
20     cout << "Test 1:\n";
21     displayGrade(test1);    // GradedActivity object
22     cout << "\nTest 2:\n";
```

```

23     displayGrade(test2);    // PassFailExam object
24     return 0;
25 }
26
27 //*****
28 // The displayGrade function displays a GradedActivity object's *
29 // numeric score and letter grade.                               *
30 //*****
31
32 void displayGrade(const GradedActivity &activity)
33 {
34     cout << setprecision(1) << fixed;
35     cout << "The activity's numeric score is "
36           << activity.getScore() << endl;
37     cout << "The activity's letter grade is "
38           << activity.getLetterGrade() << endl;
39 }

```

Program Output

Test 1:

The activity's numeric score is 88.0

The activity's letter grade is B

Test 2:

The activity's numeric score is 75.0

The activity's letter grade is P

Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer, as demonstrated in the `displayGrade` function.

Base Class Pointers

- Can define a pointer to a *base* class object
- Can assign it the address of a *derived* class object

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);  
  
cout << exam->getScore() << endl;  
cout << exam->getLetterGrade() << endl;
```

Base Class Pointers

- Base class pointers and references only know about members of the base class
 - So, you can't use a base class pointer to call a derived class function
- Redefined functions in *derived* class will be ignored unless *base* class declares the function `virtual`

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.

So, a virtual function is overridden, and a non-virtual function is redefined.

Virtual Destructors

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from.
- See Program 15-14 for an example

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:

```
virtual void Y() = 0;
```
- The `= 0` indicates a pure virtual function
- Must have no function definition in the base class

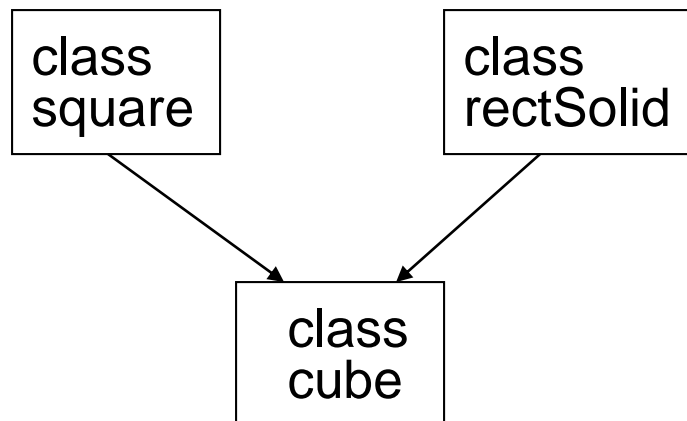
Abstract Base Classes and Pure Virtual Functions

- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function

Multiple Inheritance

- A derived class can have more than one base class
- Each base class can have its own access specification in derived class's definition:

```
class cube : public square,  
            public rectSolid;
```



Multiple Inheritance

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
 - Derived class redefines the multiply-defined function
 - Derived class invokes member function in a particular base class using scope resolution operator ::
- Compiler errors occur if derived class uses base class function without one of these solutions

WHAT IS AN INHERTANCE?

- ❑ Inheritance is the process by which new classes called derived classes are created from existing classes called base classes.
- ❑ The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.
- ❑ The idea of inheritance implements the **is a relationship**. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

WHAT IS AN INHERTANCE? contd...

MAMMAL

All mammals have certain characteristics.



Dog *is a* mammal. It has all features of mammals in addition to its own unique features



Cat *is a* mammal. It has all features of mammals in addition to its own unique features

FEATURES /ADVANTAGES OF INHERITANCE

- ❑ Reusability of Code
- ❑ Saves Time and Effort
- ❑ Faster development, easier maintenance and easy to extend
- ❑ Capable of expressing the inheritance relationship and its transitive nature which ensures closeness with real world problems .

SYNTAX

To create a derived class from an already existing base class the syntax is:

```
class derived-class: access-specifier base-class  
{  
  
...  
  
}
```

Where access specifier is one of public, protected, or private.

SYNTAX contd.....

For example, if the base class is *animals* and the derived class is *amphibians* it is specified as:

```
class animals //base class
{
    .....
};

class amphibians : public animals
{           //derived class

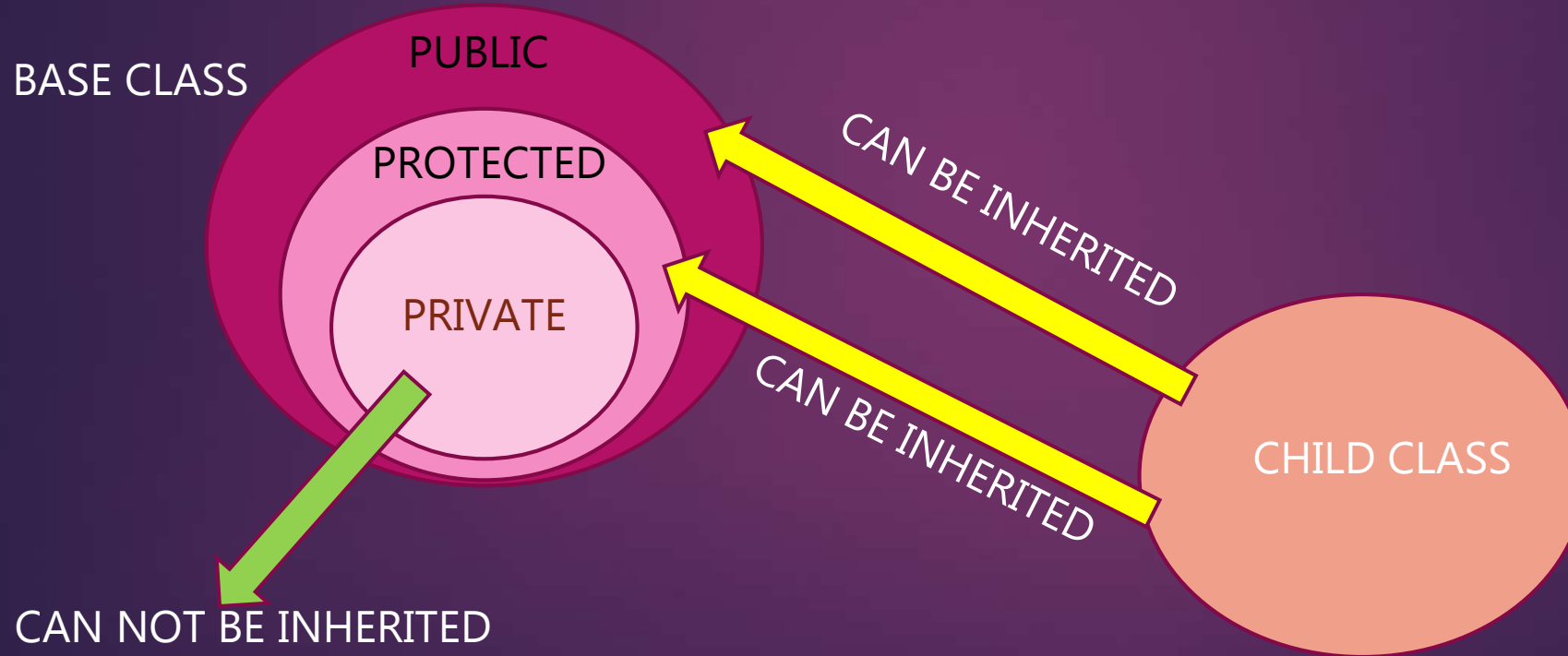
    .....
};
```

In this example class amphibians have access to both public and protected members of base class animals.

NOTE: A class can be derived from more than one class, which means it can inherit data and functions from multiple base classes. In that case a class derivation lists names of one or more base classes each separated by comma.

ACCESS CONTROL AND INHERITENCE

- ❑ A derived class can access all the protected and public members of its base class.
- ❑ It can not access private members of the base class.



ACCESS CONTROL AND INHERITENCE contd...

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

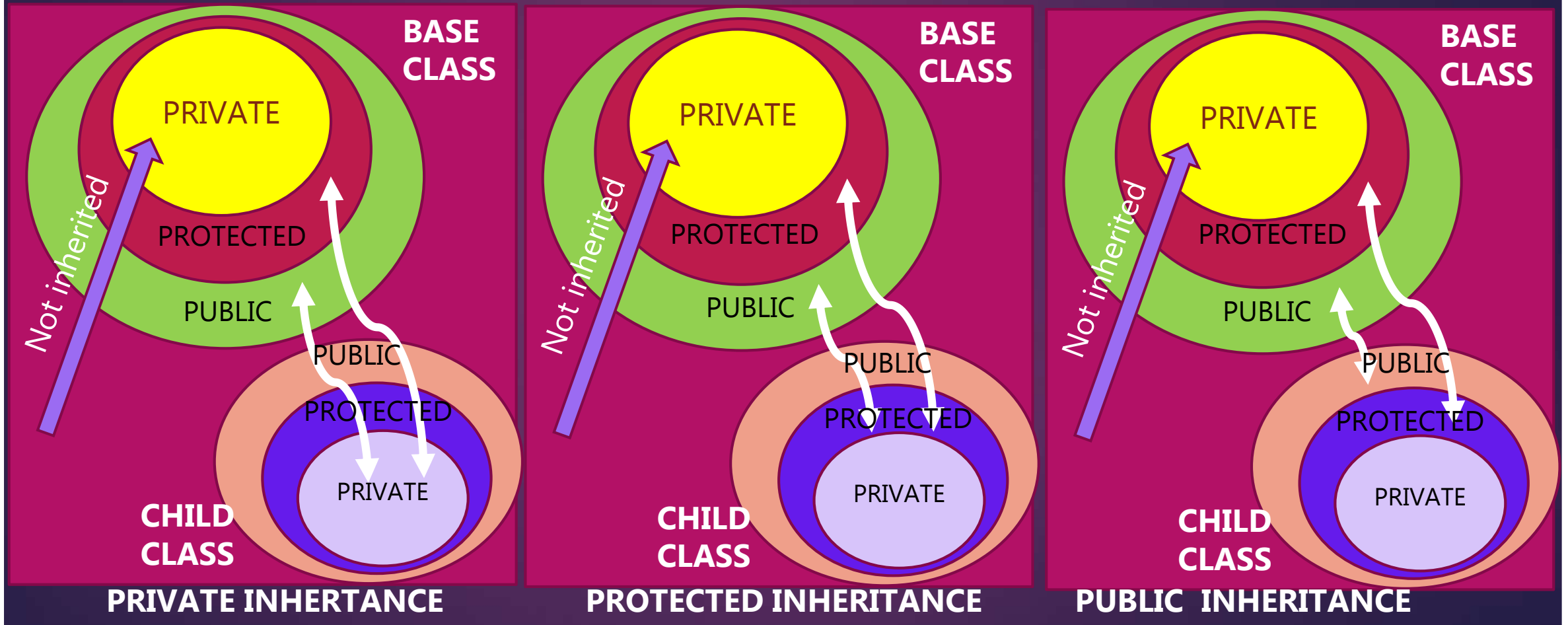
NOTE: Constructors and destructors of the base class are never inherited.

VISIBILITY MODES AND INHERITANCE

A child class can inherit base class in three ways. These are:

<i>Inheritance type</i> / <i>Members of base class</i>	PRIVATE	PROTECTED	PUBLIC
PRIVATE	NOT INHERITED	Become private of child class	Become private of child class
PROTECTED	NOT INHERITED	Become protected members of child class	Become protected members of child class
PUBLIC	NOT INHERITED	Become protected members of child class	Become public members of child class

VISIBILITY MODES AND INHERITANCE



PRIVATE INHERITANCE

In private inheritance protected and public members of the base class become the private members of the derived class.

```
class base
{
private:
int a;
void funca();
protected:
int b;
void funcb();
public:
int c;
void funcc();
}
```

Private
← inheritance

```
class child : private base
{
private:
int x;
void funcx();
protected:
int y;
void funcy();
public:
int z;
void funcz();
}
```

```
class child
{
private:
int x;
void funcx();
int b;
void funcb();
int c;
void funcc();
protected:
int y;
void funcy();
public:
int z;
void funcz();
}
```

Protected members
inherited from base
class

Public members
inherited from base
class

New child class after inheritance

PROTECTED INHERITANCE

In protected inheritance protected and public members of the base class become the protected members of the derived class.

```
class base
{
private:
int a;
void funca();
protected:
int b;
void funcb();
public:
int c;
void funcc();
}
```

Protected
Inheritance

```
class child : protected base
{
private:
int x;
void funcx();
protected:
int y;
void funcy();
public:
int z;
void funcz();
}
```

```
class child
{
private:
int x;
void funcx();
protected:
int y;
void funcy();
int b;
void funcb();
int c;
void funcc();
public:
int z;
void funcz();
}
```

Protected members
inherited from base
class

Public members
inherited from base
class

New child class after inheritance

PUBLIC INHERITANCE

In protected inheritance protected members become the protected members of the base class and public members of the base class become the public members of the derived class.

```
class base
{
private:
int a;
void funca();
protected:
int b;
void funcb();
public:
int c;
void funcc();
}
```

Public
← Inheritance

```
class child : public base
{
private:
int x;
void funcx();
protected:
int y;
void funcy();
public:
int z;
void funcz();
}
```

```
class child
{
private:
int x;
void funcx();
protected:
int y;
void funcy();
int b;
void funcb();
public:
int z;
void funcz();
int c;
void funcc();
}
```

Protected members
inherited from base
class

Public members
inherited from base
class

New child class after inheritance

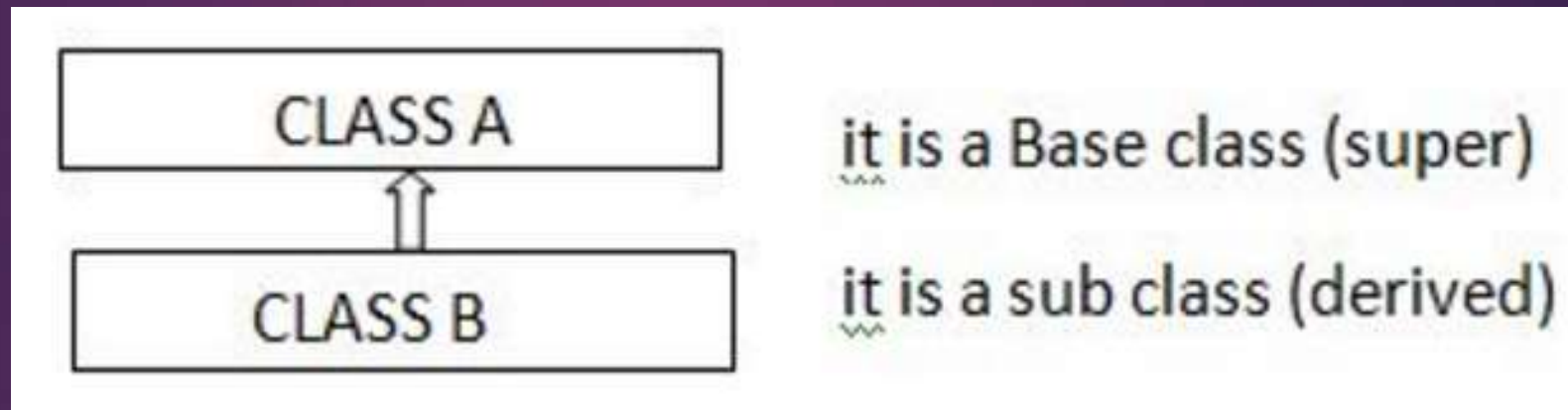
TYPES OF INHERITANCE

There are five different types of inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

SINGLE INHERITENCE

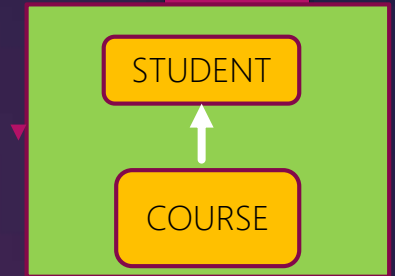
Single inheritance is the one where you have a single base class and a single derived class.



SINGLE INHERITENCE EXAMPLE

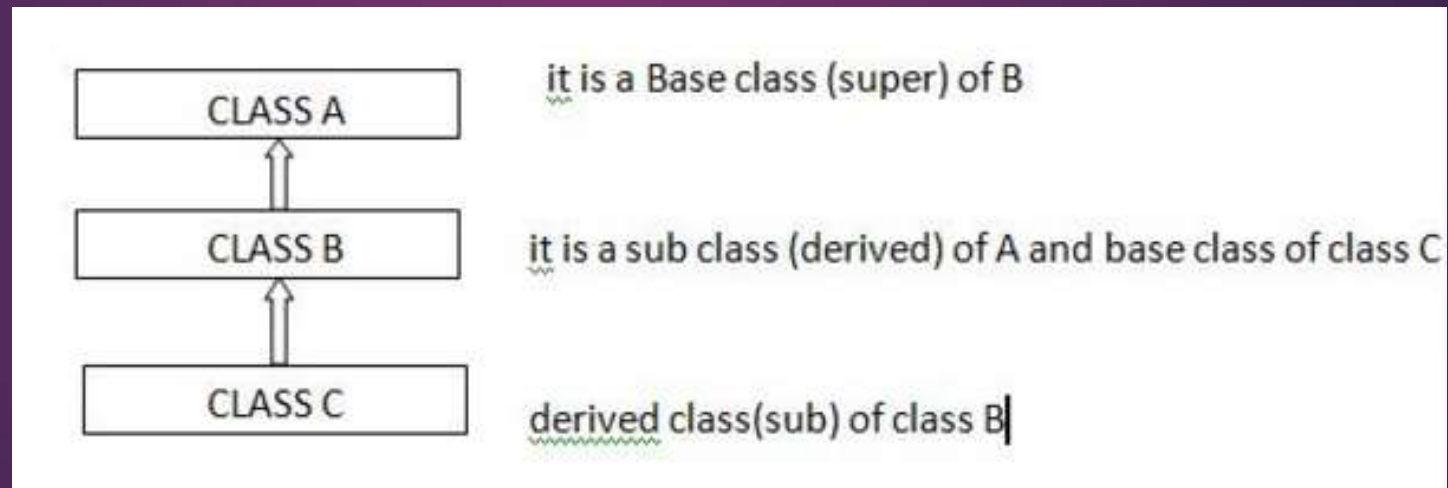
```
class student
{
private:
char name[20];
float marks;
protected:
void result();
public:
student();
void enroll();
void display();
}
```

```
class course : public student
{
long course_code;
char course_name;
public:
course();
void commence();
void cdetail();
}
```

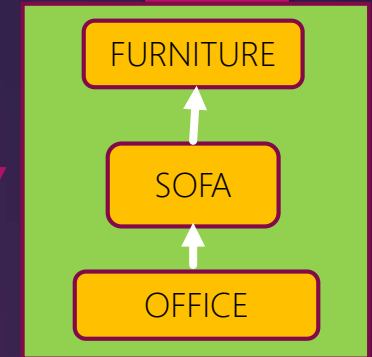


MULTILEVEL INHERITENCE

In Multi level inheritance, a subclass inherits from a class that itself inherits from another class.



MULTILEVEL INHERITENCE EXAMPLE



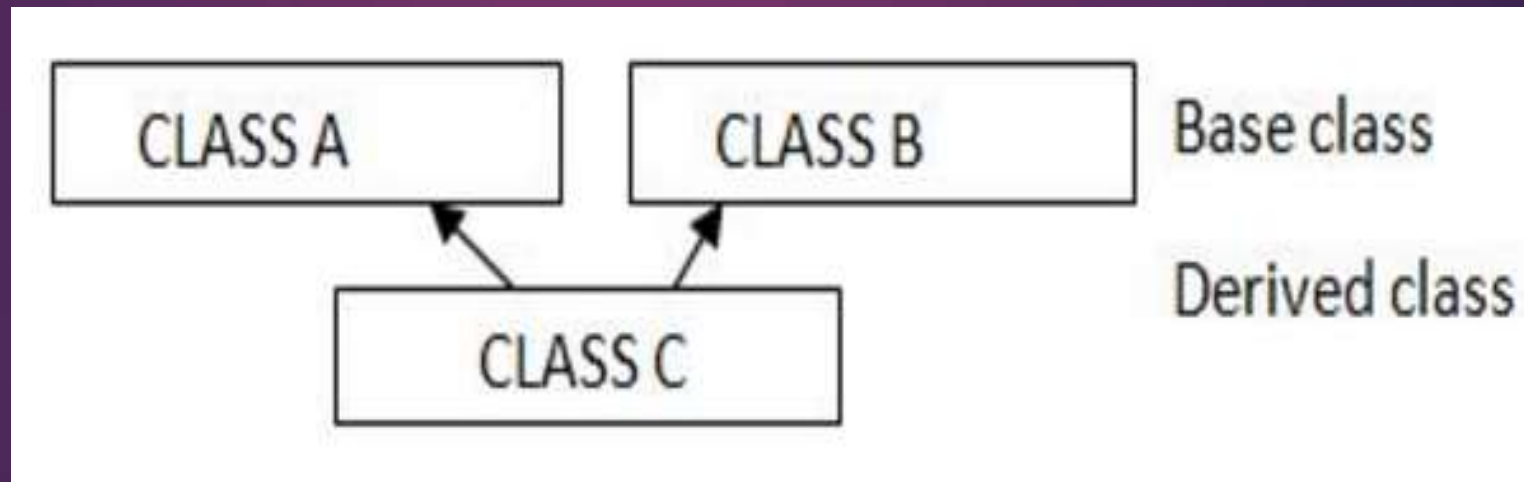
```
class furniture
{
char type;
char model[10];
public:
furniture();
void readdata();
void dispdata();
}
```

```
class sofa: public furniture
{
int no_of_seats;
float cost;
public:
void indata();
void outdata();
};
```

```
class office: private sofa
{
int no_of_pieces;
char delivery_date[10];
public:
void readdetails()
void displaydetails();
}
```

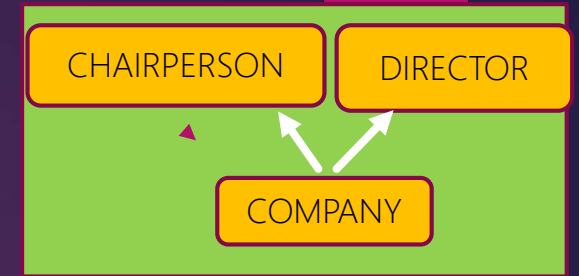
MULTIPLE INHERITENCE

In Multiple inheritances, a derived class inherits from multiple base classes. It has properties of both the base classes.



MULTIPLE INHERITENCE

EXAMPLE



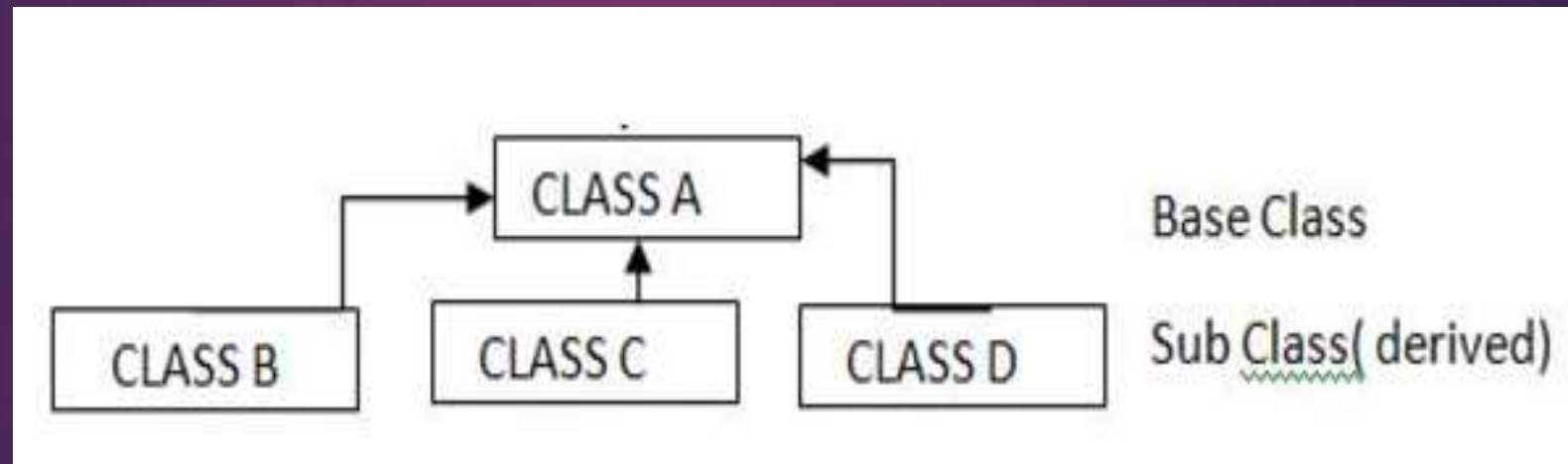
```
class chaiperson
{
long chairid;
char name[20];
protected:
char description[20];
void allocate();
public:
chairperson();
void assign();
void show();
};
```

```
class director
{
long directorid;
char dname[20];
public:
director();
void entry();
void display();
};
```

```
class company: private
chairperson, public director
{
int companyid;
char city[20];
char country[20];
public:
void ctenry();
void ctdisplay();
};
```

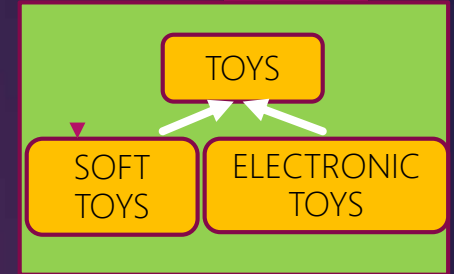
HIERARCHICAL INHERITENCE

In hierarchical Inheritance, it's like an inverted tree. So multiple classes inherit from a single base class.



HIERARCHICAL INHERITENCE

EXAMPLE



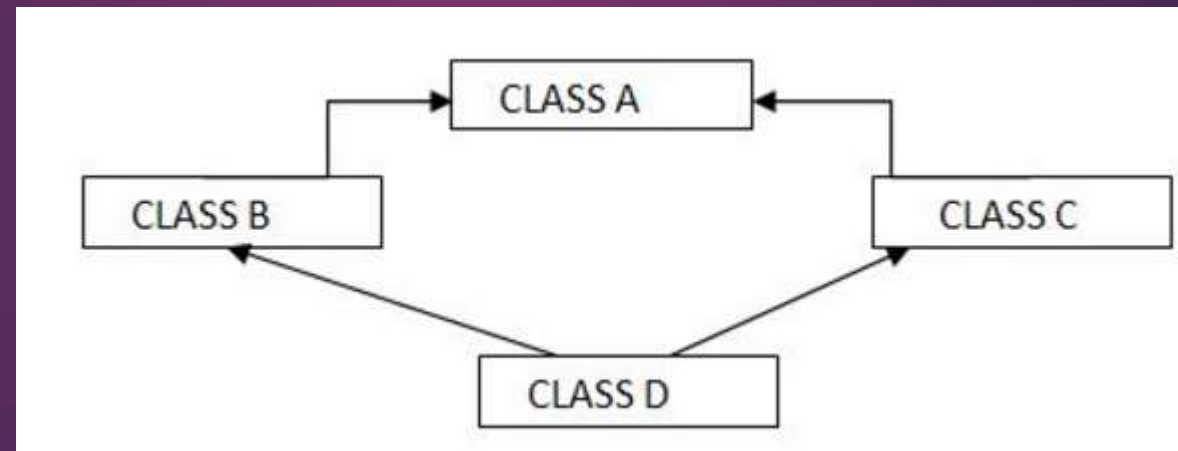
```
class toys
{
char tcode[5];
protected:
float price;
void assign(float);
public:
toys();
void tentry();
void tdisplay();
};
```

```
class softtoys: public toys
{
char stname[20];
float weight;
public:
softtoys();
void stentry();
void stdisplay();
};
```

```
class electronictoys: public
toys
{
char etname[20];
int no_of_batteries;
public:
void etentry();
void etdisplay();
};
```

HYBRID INHERITENCE

It combines two or more types of inheritance. In this type of inheritance we can have a mixture of number of inheritances.



CONSTRUCTORS AND DESTRUCTORS IN BASE AND DERIVED CLASSES

- ❑ Derived classes can have their own constructors and destructors.
- ❑ When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor.
- ❑ When an object of a derived class goes out of scope, its destructor is called first, then that of the base class.

IMPOTANT POINTS TO NOTE

❑ Calculating the size of the object of the child class:

- ❖ While calculating the size of the object of the child class, add the size of all data members of base class including the private members of the base class and the child class.
- ❖ If child class is inheriting from multiple base classes, add the size of data members of all base classes and the child class.
- ❖ In case of multilevel inheritance the size of all base classes(directly /indirectly) inherited by child class is added to the size of child class data members

❑ Members accessible to the object of the child class:

Only public members of the new modified child class(after inheritance) are accessible to the object of the child class.

❑ Members accessible to the functions of the child class:

All members: public, protected, private, of the new modified child class(after inheritance) are accessible to the functions of the child class.

PASSING ARGUMENTS TO BASE CLASS CONSTRUCTOR

If a base class has parametrized constructor then it is the duty of child class to pass the parameters for base class constructor also at the time of creation of object.

```
class student
{
private:
char name[20];
float marks;
protected:
void result();
public:
student(char nam[20], float mar);
void enroll();
void display();
}
```

Base class constructor

```
class course : public student
{
long course_code;
char course_name[20];
public:
course(long cc, char cn[20],char nam[20], float mar ) :
student(char nam[20], float mar);
void commence();
void cdetail();
}
course c1(01,"CS","Naman", 460);
```

Child class constructor

Base class constructor parameters