Lecture 6

Inheritance

Outline

- Reusability
- Base class and derived class
- Function Overriding
- Access Control and Inheritance
- Public Inheritance
- Constructors and Destructors in Inheritance
- Assignment Operator and Inheritance
- Multiple Inheritance
- Virtual Base Classes
- The typeid function

Reusability in Object-Oriented Programming

- Reusability means taking an existing class and using it in a new program.
- By reusing classes, you can reduce the time and effort needed to develop a program.

- The simplest way to reuse a class is to just use an object of that class directly.
  - The standard library of the C++ has many classes.
  - Example: The built-in string class.

- The second way to reuse a class is to place an object of that class inside a new class.
  - The new class contains objects of existing classes.
  - This concept is called composition (**Has-a**).

Reusability in Object-Oriented Programming

The third way to reuse a class is **inheritance**.

- Inheritance is one of the ways in object-oriented programming that makes reusability possible.

- By the help of inheritance we can write more special classes from general classes.

- Inheritance relation is referred to as **(Is-a)** or **(A-kind-of)** .

Example : Using built-in string class

- To use built-in strings, the C++ header file **<string>** is included.
- Because of built-in operator overloadings, syntax for using strings is easy.

```
#include <string>              // String class header file
#include <iostream>
using namespace std;

int main()
{
   string  s1 = "ABC";          // Initialized by assignment
   string  s2 ("DEF");          // Initialized by constructor
   string  s3;                  // Empty string

   s3  =  s1  +  " , "  +  s2;  // Combining strings (overloaded +)
   s3  +=  "XYZ";               // Appending a string (overloaded +=)

   cout << s3 << endl;
}
```

Inheritance

- OOP provides a way to modify a class without changing its code.
- This is achieved by using **inheritance** to derive a new class from the old one.
- The old class (called the **base class**) is not modified, but the new class (the **derived class**) can use all the features of the old one and additional features of its own.

Generalization and Specialization

- By the help of inheritance we can write more special classes from general classes.
- Special classes may have more members (data and functions) than general classes.

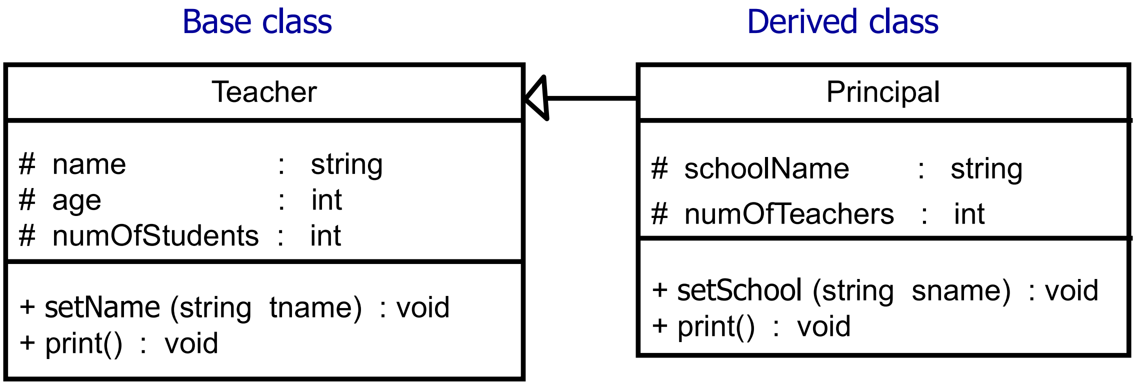
Employee → Worker → Manager

Employee is base class (general class)
Worker is-a-kind of Employee (special class)
Manager is-a-kind of Worker (special class)

Vehicle → Air vehicle → Helicopter

Vehicle is base class (general)
Air vehicle is derived class (special)
Helicopter is derived class (special)

Example: UML Class Diagram for Inheritance



Base class

Derived class

Teacher

# name : string
# age : int
# numOfStudents : int

+ setName (string tname) : void
+ print() : void

Principal

# schoolName : string
# numOfTeachers : int

+ setSchool (string sname) : void
+ print() : void

Access specifiers
# is protected
+ is public

Example : Base class

- Defining classes Teacher and Principal (director) in a school.
- First, we define the Teacher class.
- Then we can use Teacher class to define the Principal class.
- Protected members of base class will be public in derived class members.

```
class Teacher    // Base class
{
  protected:
    string  name;
    int       age;
    int       numOfStudents;

   public:
     void  setName (string  tname)
     {
         name = tname;
     }
     void  print ();
};
```

Example : Derived class

Principal is a special type of Teacher. It has more member data and functions.

```
class  Principal  :  public  Teacher   // Derived class
{
    string  schoolName;      // Additional members
    int      numOfTeachers;

  public:
    void  setSchool  (string   sname)
          { schoolName  =  sname; }
    void  print ();
};
```

```
int main()
{
   Teacher    t1;
   Principal   p1;

   t1 . setName ("Teacher1");
   p1 . setName ("Principal1");
   p1 . setSchool ("School1");
}
```

Base and Derived Classes

- An object of a **derived class** inherits all the member data and functions of the **base class**.
- **Private** members of the base class are inherited by the derived class, but they are not visible in the derived class.
- The member functions of the derived class can not access private members of the base class directly.
- The derived class may access them only through the public interface functions of the base class.

Redefining Members (Overriding)

- Sometimes member data and functions should be redefined in the derived class.
- Example : The print function must be redefined in Principal class, because principals have more properties to print.

```
class Teacher      // Base class
{
  protected:
    string  name;
    int  age, numOfStudents;

  public:
    void  setName (string  tname) { name = tname; }
    void  print ();
};
```

```
void  Teacher :: print ()   // Print function of Teacher class
{
   cout << "Name: " << name<< "  Age: " << age << endl;
   cout << "Number of Students: " << numOfStudents << endl;
}
```

- The print() function of the Principal class **overrides** (hides) the print() function of the Teacher class.
- The Principal class has actually two print() functions. (The inherited print function and its own print function.)

```
class Principal : public Teacher    // Derived class
{
     string  schoolName;
     int  numOfTeachers;

   public:
     void setSchool (string  sname) { schoolName = sname; }
     void print() ;    // Print function of Principal class (Overridden)
};
```

```
void  Principal :: print()    // Print function of Principal class (Overridden)
{
    cout << "Name: " << name << "  Age: " << age << endl;
    cout << "Number of Students: " << numOfStudents << endl;
    cout << "Name of the school: " <<  schoolName << endl;
}
```

Alternative method:

- Members (data and function) of the base class can be accessed by using the scope operator (::) .
- We can call print function of base class, in print function of derived class.

```
void  Principal :: print()     // Print function of Principal class (Overridden)
{
    Teacher :: print();    // Invokes print function of Teacher class

    cout << "Name of the school: " << schoolName << endl;
}
```

Overriding the member functions

- If the author of the **derived class** redefines a member function, it means he changes the interface of the **base class**.
- In this case the member function of the base class is hidden (**overridden**).
- The hidden members in base class are still accessible through the scope :: operator.
- The derived class will have **two member functions** with the same name.

Example: Overriding of member data and functions

```
class A
{
 public:
   int    i1, i2;
   void  f1 ();
   int    f2 (int);
};
```

```
class B: public A
{
 public:
   float  i1;            // overrides  i1
   float  f1 (float);    // overrides  f1
};
```

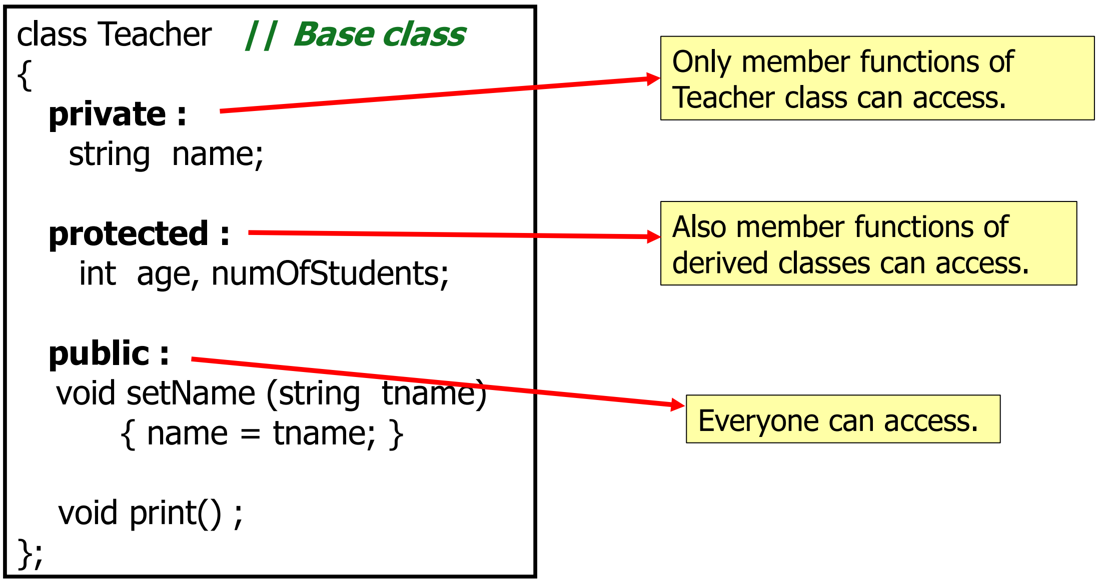
```
int main() {
  B   b;
  int   j      =   b . f2 (10);        // A :: f2
  b . i1      =   40;                   // B :: i1
  b . i2      =   30;                   // A :: i2, because i2 is public in A
  float   y  =   b . f1 (3.14);       // B :: f1
  b . f1 ();          // Compile error  f1(float) in B hides the f1(void) of A
  b . A :: f1 ();                     // OK, because f1(void) is public in A
  b . A :: i1  =  20;                 // OK, because i1 is public in A
}
```

Access Control and Inheritance

- When inheritance is not involved, class member functions have access to anything in the class, whether public or private. But objects of that class have access only to public members.
- Member functions of a derived class can access **public** and **protected** members of base class, but not **private** members.
- Objects of a derived class can access only public members of base class.

Access specifier	Accessible from own class	Accessible from derived class	Accessible from other objects or from main
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	Yes	No	No

Example : Access Specifiers



```
class Teacher   // Base class
{
   private :
     string  name;

   protected :
      int  age, numOfStudents;

   public :
    void setName (string  tname)
          { name = tname; }

    void print() ;
};
```

Only member functions of Teacher class can access.

Also member functions of derived classes can access.

Everyone can access.

```
class Principal : public Teacher  // Derived class
{
  private:                          // Default access specifier
    string  schoolName;
    int      numOfTeachers;

  public:
    void  setSchool (string  sname)  { schoolName = sname; }
    void  print();
    int  getAge()  { return age; }   // It works because age is protected
    string  get_name() { return name;}  //ERROR (because name is private)
};
```

```
int main()  {
    Teacher    T1;
    Principal    P1;
    T1 . numOfStudents  =  100;
    //Error  (numOfStudents  is  protected)

    T1 . setName ("Teacher1");
    P1 . setSchool ("School1");
}
```

Public Inheritance

- The access specifier of derivation is usually written as public.
- In **public inheritance**, the objects of derived class can access public members of base class.
- Public members of base class are also public members of derived class.

```
class Base { …… };

class Derived : public Base { …… };
```

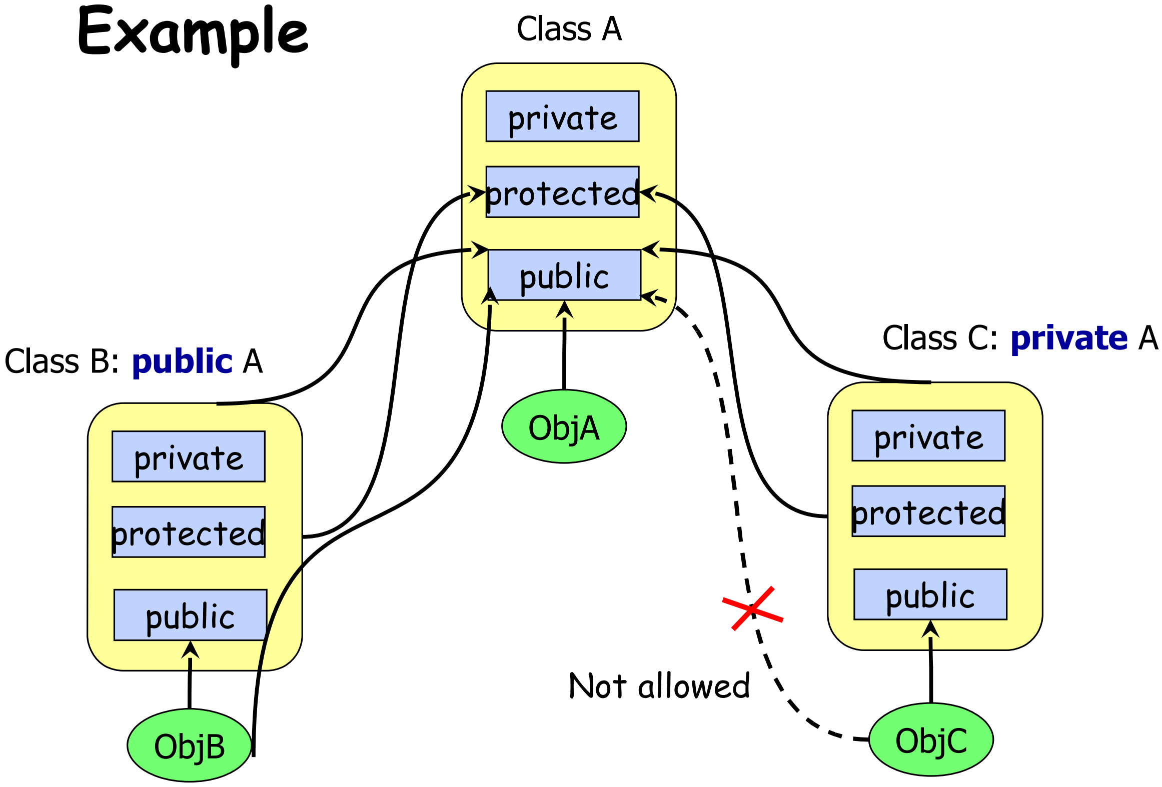
Private Inheritance

- In **private inheritance**, the public members of base class are private members of derived class.
- Objects of derived class can not access any members of base class.
- Member functions of derived class can still access public and protected members of base class.

```
class Base { …… };

class Derived : private Base { …… };
```

Example



Class A

private

protected

public

ObjA

Class B: **public** A

private

protected

public

ObjB

Class C: **private** A

private

protected

public

ObjC

Not allowed

Special Member Functions and Inheritance

- Some special functions will need to do different things in the base class and in the derived class.

- **Overloaded assignment operator:**
  The = operator in the derived class must assign values to derived class data, and the = operator in the base class must assign values to base class data.

- **Constructors** :
  Because the derived class and base class constructors initialize different data, one constructor cannot be used in place of another.

Constructors and Inheritance

- When you define an object of a derived class, the base class constructor will be automatically invoked (called) before the derived class constructor.

```
class Teacher {   //Base class
{
   string  name;
   int  age, numOfStudents;
  public:

   // Constructor of base class
   Teacher (string  tname)  :  name (tname) { }
};
```

```
class  Principal :  public  Teacher  // Derived class
{
   int  numOfTeachers;
  public:
   // Constructor of derived class
   Principal (string , int );
};
```

- If the base class has a constructor, which must take some arguments, then the derived class must also have a constructor that calls the constructor of the base with proper arguments.

```
// Constructor of derived class
Principal :: Principal  (string  tname,  int numOT)  :  Teacher (tname)
{
    numOfTeachers  =  numOT;
}
```

Base constructor

- Constructor of base class **can not be called directly** as a statement in function block.
- It can be called only in constructor initializer list.

```
Principal : : Principal  (string  tname,  int  numOT)
{
    numOfTeachers  =  numOT;
    Teacher (tname) ;  // Compiler error (Base constructor)
}
```

Destructors and Inheritance

- When an object of derived class goes out of scope, the destructors are called in reverse order.
- The derived object is automatically destroyed first, then the base class object is destroyed.

```
class  B     // Base class
{
  public:
     B()    { cout << "B constructor" << endl; }
    ~B()   { cout << "B destructor" << endl; }
};
```

```
class  C  :  public  B    // Derived class
{
  public:
     C()    { cout << "C constructor" << endl; }
     ~C()  { cout << "C destructor" << endl; }
};
```

```
int  main ()
{
   cout << "Program started \n";

   C   x ;     // Object of derived class

   cout << " Program ended \n";
}
```

Screen output

```
Program started
B constructor
C constructor

Program ended
C destructor
B destructor
```

Overloaded Assignment Operator and Inheritance

In inheritance, assignment operator of derived class should be different than assignment operator of base class.

```
class  String   //Base class
{
  protected:
     int  size;
     char  *contents;
  public:
     const  String  &  operator=  (const  String &);
     // Overloaded assignment operator
};
```

```
const  String  &   String :: operator=  (const  String &   in_object)
{
   delete [ ]  contents;         // Delete old contents
   size  =  in_object.size;
   contents  =  new char[size+1];
   strcpy (contents,  in_object . contents);
   return  *this;
}
```

- String2 class is derived from String class.
- It has two contents.
- First is inherited from base class, second is declared as an extra.

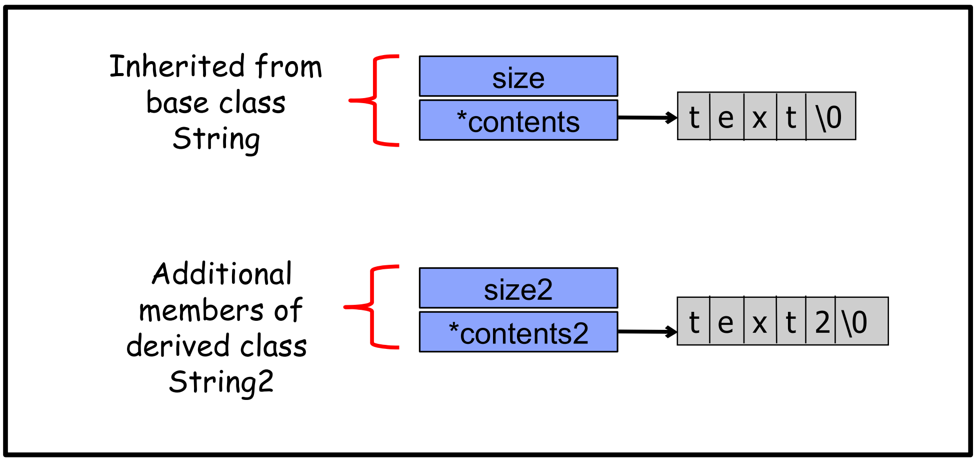
```
class  String2  :  public  String   // Derived class
{
  int  size2;
  char  *contents2;

                                 contents          contents2
 public:
  String2 ();                                         // Default constructor
  String2 (const  char * , const char *);    // Parametered constructor
  String2 (const  String2  &);                 // Copy constructor

  const  String2 &   operator=  (const  String2  &);
   // Overloaded assignment operator

  void  print()  const;
  ~String2 ();
};
```

Members of derived class String2


Inherited from base class String
size
*contents
t e x t \0
Additional members of derived class String2
size2
*contents2
t e x t 2 \0

Overloaded assignment operator of String2 class

- Data members of String class (base) must be protected.
- Otherwise functions of String2 class (derived) can not access them.

```
const  String2 &    String2 :: operator=   (const  String2   &in_object)
{
   // Copy the inherited data members (base class)
   size = in_object.size;
   delete [ ] contents;
   contents = new  char[size + 1];
   strcpy (contents,  in_object.contents);

   // Copy the additional data members (derived class)
   size2  =  in_object . size2;
   delete [ ]  contents2;
   contents2  =  new  char[size2 + 1];
   strcpy (contents2,  in_object . contents2);

   return  *this;
}
```

Inheritance and Composition

- **Inheritance** represents the **is-a** relation.
  Example: Class B is-a kind of class A.

- **Composition** represents the **has-a** relation.
  Example: Class C has-a class A object.

```
class  A  { };
```

```
class  B  :  public  A {};
```

B is-a A (Inheritance)

```
class  C
{
   A   a;
};
```

C has-a A (Composition)

Example : Inheritance and Composition

Inheritance and composition can be used together.

```
class A
{
  public:
  void  f () {cout << "A is called \n" ;}
};
```

```
class B
{
  public:
  void  f () {cout << "B is called \n";}
};
```

```
class  C  :  public  B      // Inheritance, C  is-a  B
{
  A  a;                     // Composition, C  has-a  A

  public:
  void  f ()             // Redefinition (function override)
  {
       a . f ();         // Call function f  of  a
       B :: f ();        // Call function f  of base class  B
  }
};
```

```
int main()
{
  C   c;
  c . f ();
}
```

Screen output

```
A is called
B is called
```

Multiple Inheritance

Multiple inheritance occurs when a class inherits from two or more base classes.

```
class  Base1
{
   public:
   void    f1 ()
   char*  f2 (int);
};
```

```
class  Derived  :  public  Base1 ,  public  Base2
{
    public:
    float  f1 (float);   // override Base1
    void  f4 ();         // override Base2
    int    f5 (int);
};
```

```
class  Base2
{
   public:
   char*  f2 (int, char);
   int     f3 ();
   void    f4 ();
};
```

```
int main()
{
  Derived  d;
  float  y  =  d . f1 (0.8);   // Derived :: f1
  d . f3 ();                   // Base2 :: f3
  d . f4 ();                   // Derived :: f4
  d . Base2 :: f4 ();          // Base2 :: f4
}
```

The Derived class contains the following 8 member functions.

Base1 :: f1

Base1 :: f2

Base2 :: f2

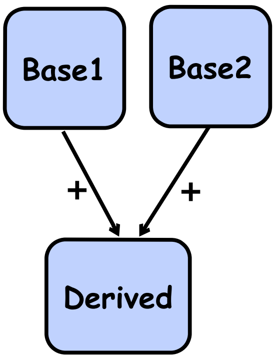
Base2 :: f3

Base2 :: f4

Derived :: f1

Derived :: f4

Derived :: f5



Repeated Base Classes

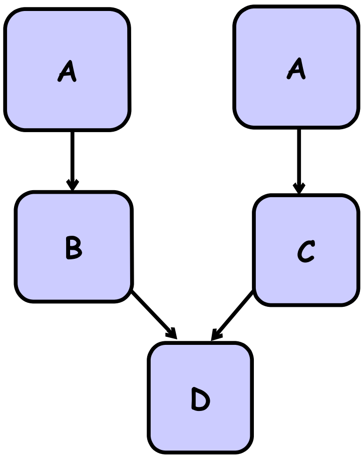
- In the following example, D object will contain two A subobjects, one inherited via B and one inherited via C.
- This is a repeated (duplicated) inheritance.
- There are two subobjects when really there should be only one.

```
class A   {   };

class B : public A    {   };

class C : public A    {   };

class D : public B, public C    {   };
```



Virtual Base Classes

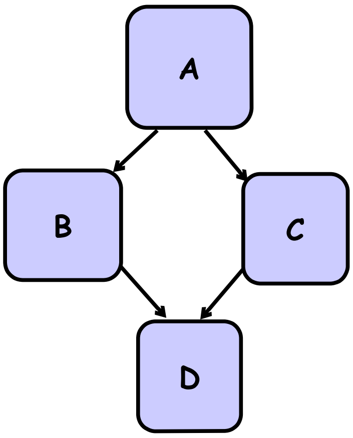
- To fix repeated inheritance, the **virtual** keyword can be used, when deriving B and C from A.
- The **virtual** keyword tells the compiler to inherit only one A subobject from base class A.

```
class A    {   };

class B : virtual public A   {   };

class C : virtual public A   {   };

class D : public B, public C   {   };
```



Using built-in typeid Function

- In <typeinfo> header file, there is a built-in function called typeid.
- The typeid function returns the type info object about a class.
- The name() member function of type info object returns a string containing the user-written class name.

```
#include <iostream>
#include <typeinfo>
using namespace std;

class Taban { };
class Turetilen : public Taban { };

int main() {
Taban       a;
Turetilen   b;
const  type_info&  id1 = typeid (a);
const  type_info&  id2 = typeid (b);

cout << id1 . name() << endl;
cout << id2 . name() << endl;
// Alternative calling:
cout << typeid (a) . name () << endl;
}
```

Visual C++ Screen output

```
class Taban
class Turetilen
```

("class" word is shown before the class name.)

Dev-C++ Screen output

```
5Taban
9Turetilen
```

(Number of characters in class name is shown before the class name.)