ACA Summer School 2014
Advanced C++

Pankaj Prateek

ACA, CSE, IIT Kanpur

June 25, 2014

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer while debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer while debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer while debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer while debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer while debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer while debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer while debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer while debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Inheritance

```
class new_class : visibility_mode old_class {
  // Some stuff for new class
  // Nothing special
};
```

- Indicates that the new_cass had been derived from the old_class in the specified visibility mode.

Inheritance

```
class new_class : visibility_mode old_class {
  // Some stuff for new class
  // Nothing special
};
```

- Indicates that the new_cass had been derived from the old_class in the specified visibility mode.

Inheritance

Only “public” members are inherited. Private members of base class are never accessible to any derived class. They can be accessed indirectly using public members from the same base class.

Inheritance: Private

```
class XYZ {
  // members of XYZ
}
class ABC : private XYZ {
  // members of ABC
}
```

- Privately inherited/derived: “Public” members of the base class become “private” members of the derived class
- Thus the public members of XYZ inherited into ABC can only be accessed through public functions of ABC
- As a result, the inherited members of the base class are not directly accessible to the objects of the derived class. Access is restricted to the public members only

Inheritance: Private

```
class XYZ {
  // members of XYZ
}
class ABC : private XYZ {
  // members of ABC
}
```

- Privately inherited/derived: “Public” members of the base class become “private” members of the derived class
- Thus the public members of XYZ inherited into ABC can only be accessed through public functions of ABC
- As a result, the inherited members of the base class are not directly accessible to the objects of the derived class. Access is restricted to the public members only

Inheritance: Private

```
class XYZ {
  // members of XYZ
}
class ABC : private XYZ {
  // members of ABC
}
```

- Privately inherited/derived: “Public” members of the base class become “private” members of the derived class
- Thus the public members of XYZ inherited into ABC can only be accessed through public functions of ABC
- As a result, the inherited members of the base class are not directly accessible to the objects of the derived class. Access is restricted to the public members only

Inheritance: Private

```
class XYZ {
  // members of XYZ
}
class ABC : private XYZ {
  // members of ABC
}
```

- Privately inherited/derived: “Public” members of the base class become “private” members of the derived class
- Thus the public members of XYZ inherited into ABC can only be accessed through public functions of ABC
- As a result, the inherited members of the base class are not directly accessible to the objects of the derived class. Access is restricted to the public members only

Inheritance: Private

```
// Derived class in private Inheritance
// How it looks in concept
// NOT REAL!!!
class ABC: private XYZ {
private:
  ... private members of XYZ
  ... private members of ABC
public:
  ... public members of ABC
};
```

Inheritance: Public

```
class XYZ {
  // members of XYZ
}
class ABC : public XYZ {
  // members of ABC
}
```

- Publicaly inherited/derived: “Public” members of the base class become “public” members of the derived class
- Thus the public members of XYZ inherited into ABC can be accessed directly by the objects of ABC

Inheritance: Public

```
class XYZ {
  // members of XYZ
}
class ABC : public XYZ {
  // members of ABC
}
```

- Publicaly inherited/derived: “Public” members of the base class become “public” members of the derived class
- Thus the public members of XYZ inherited into ABC can be accessed directly by the objects of ABC

Inheritance: Public

```
class XYZ {
  // members of XYZ
}
class ABC : public XYZ {
  // members of ABC
}
```

- Publicaly inherited/derived: “Public” members of the base class become “public” members of the derived class
- Thus the public members of XYZ inherited into ABC can be accessed directly by the objects of ABC

Inheritance: Public

```
// Derived class in private Inheritance
// How it looks in concept
// NOT REAL!!!
class ABC: private XYZ {
private:
  ... private members of ABC
public:
  ... public members of XYZ
  ... public members of ABC
};
```

Inheritance: Protected

- At times we may require that a private member be inherited by the derived class
- An obvious way to achieve this is to make it public. But at the cost of losing the advantage of Data Hiding
- Hence to facilitate this, C++ has a new keyword : protected
- A “protected” member of a class, is accessible by the member functions within that class or any class “immediately” derived from it

Inheritance: Protected

- At times we may require that a private member be inherited by the derived class
- An obvious way to achieve this is to make it public. But at the cost of losing the advantage of Data Hiding
- Hence to facilitate this, C++ has a new keyword : protected
- A "protected" member of a class, is accessible by the member functions within that class or any class "immediately" derived from it

Inheritance: Protected

- At times we may require that a private member be inherited by the derived class
- An obvious way to achieve this is to make it public. But at the cost of losing the advantage of Data Hiding
- Hence to facilitate this, C++ has a new keyword : protected
- A "protected" member of a class, is accessible by the member functions within that class or any class "immediately" derived from it

Inheritance: Protected

- At times we may require that a private member be inherited by the derived class
- An obvious way to achieve this is to make it public. But at the cost of losing the advantage of Data Hiding
- Hence to facilitate this, C++ has a new keyword : protected
- A "protected" member of a class, is accessible by the member functions within that class or any class "immediately" derived from it

Inheritance: Protected

```
class class_name {
private:
  // visible only to member functions
  // of this class
protected:
  // visible only to member functions of
  // this class and immediate derived class
public:
  // visible to all functions in the program
}
```

Inheritance: Protected

- The keyword protected is used in the same way public and private are used
- With this, C++ also allows a protected mode of inheritance as well along with public and private
- Private and protected members of a class can now be accessed by friend functions or member functions of a friend class

Inheritance: Protected

- The keyword protected is used in the same way public and private are used
- With this, C++ also allows a protected mode of inheritance as well along with public and private
- Private and protected members of a class can now be accessed by friend functions or member functions of a friend class

Inheritance: Protected

- The keyword protected is used in the same way public and private are used
- With this, C++ also allows a protected mode of inheritance as well along with public and private
- Private and protected members of a class can now be accessed by friend functions or member functions of a friend class

Visibility of Inherited Members

Table: Derived Class Members: Visibility

Declaration →	Private	Protected	Public
private	Not inherited	Not inherited	Not inherited
protected	Private	Protected	Protected
public	private	protected	public

Multilevel Inheritance

- It is when there is a series of inheritance from one class to its child class
- The general rules of inheritance are followed as they are.

$$\begin{tikzcd}
\text{Class A} \\
\text{Class B} \arrow[u] \\
\text{Class C} \arrow[u]
\end{tikzcd}$$

- Class B is also called intermediate base class of A
- The chain A-B-C is called inheritance path

Multilevel Inheritance

- It is when there is a series of inheritance from one class to its child class
- The general rules of inheritance are followed as they are.

$$\begin{tikzcd}
\text{Class A} \\
\text{Class B} \arrow[u] \\
\text{Class C} \arrow[u]
\end{tikzcd}$$

- Class B is also called intermediate base class of A
- The chain A-B-C is called inheritance path

Multilevel Inheritance

- It is when there is a series of inheritance from one class to its child class
- The general rules of inheritance are followed as they are.

$$\begin{tikzcd}
\text{Class A} \\
\text{Class B} \arrow[u] \\
\text{Class C} \arrow[u]
\end{tikzcd}$$

- Class B is also called intermediate base class of A
- The chain A-B-C is called inheritance path

Multilevel Inheritance

- It is when there is a series of inheritance from one class to its child class
- The general rules of inheritance are followed as they are.

$$\begin{tikzcd}\text{Class A} \\ \text{Class B} \arrow[u] \\ \text{Class C} \arrow[u]\end{tikzcd}$$

- Class B is also called intermediate base class of A
- The chain A-B-C is called inheritance path

Multilevel Inheritance

- It is when there is a series of inheritance from one class to its child class
- The general rules of inheritance are followed as they are.

$$\begin{tikzcd}\text{Class A} \\ \text{Class B} \arrow[u] \\ \text{Class C} \arrow[u]\end{tikzcd}$$

- Class B is also called intermediate base class of A
- The chain A-B-C is called inheritance path

Multiple Inheritance

- A class can inherit from multiple classes. Hence called multiple inheritance
- Allows the user to combine several features from different classes at the beginning of creating a new class

```
class D: visibility B1, visibility B2 {
  // definition of D
};
```

- Visibility can be either public, private or protected.
- This feature is quite unique to C++. Even JAVA doesn't support multiple inheritance.

Multiple Inheritance

- A class can inherit from multiple classes. Hence called multiple inheritance
- Allows the user to combine several features from different classes at the beginning of creating a new class

```
class D: visibility B1, visibility B2 {
  // definition of D
};
```

- Visibility can be either public, private or protected.
- This feature is quite unique to C++. Even JAVA doesn't support multiple inheritance.

Multiple Inheritance

- A class can inherit from multiple classes. Hence called multiple inheritance
- Allows the user to combine several features from different classes at the beginning of creating a new class

```
class D: visibility B1, visibility B2 {
  // definition of D
};
```

- Visibility can be either public, private or protected.
- This feature is quite unique to C++. Even JAVA doesn't support multiple inheritance.

Multiple Inheritance

- A class can inherit from multiple classes. Hence called multiple inheritance
- Allows the user to combine several features from different classes at the beginning of creating a new class

```
class D: visibility B1, visibility B2 {
  // definition of D
};
```

- Visibility can be either public, private or protected.
- This feature is quite unique to C++. Even JAVA doesn't support multiple inheritance.

Multiple Inheritance

- A class can inherit from multiple classes. Hence called multiple inheritance
- Allows the user to combine several features from different classes at the beginning of creating a new class

```
class D: visibility B1, visibility B2 {
  // definition of D
};
```

- Visibility can be either public, private or protected.
- This feature is quite unique to C++. Even JAVA doesn't support multiple inheritance.

Multiple Inheritance

```
class B1 {
public:
  int abc;
}
class B2 {
public:
  int abc;
}
class D: visibility B1, visibility B2 {
  // definition of D
};
```

Class D will have two variables with the same name. Ambiguity!!!

Multiple Inheritance

```
class B1 {
public:
  int abc;
}
class B2 {
public:
  int abc;
}
class D: visibility B1, visibility B2 {
  // definition of D
};
```

Class D will have two variables with the same name. Ambiguity!!!

Multiple Inheritance: Ambiguity Resolution

- The ambiguity is only flagged as an error if you use the ambiguous member name.
- You can resolve ambiguity by qualifying a member with its class name using the scope resolution (::) operator.

Multiple Inheritance: Ambiguity Resolution

- The ambiguity is only flagged as an error if you use the ambiguous member name.
- You can resolve ambiguity by qualifying a member with its class name using the scope resolution (::) operator.

Hierarchical Design

- In the real world, the systems to be developed are far too complex in nature, and involve a lot of programming
- Professional programming is done in stages
- Beginning from the basic functionality, going up to the most advanced one
- Code is written in the form of small modules
- Sequences of inheritance starting from the most basic to the most advanced are slowly developed
- Benefits : Modular design, ease of maintenance, clarity of thought, faster error localization

Hierarchical Design

- In the real world, the systems to be developed are far too complex in nature, and involve a lot of programming
- Professional programming is done in stages
- Beginning from the basic functionality, going up to the most advanced one
- Code is written in the form of small modules
- Sequences of inheritance starting from the most basic to the most advanced are slowly developed
- Benefits : Modular design, ease of maintenance, clarity of thought, faster error localization

Hierarchical Design

- In the real world, the systems to be developed are far too complex in nature, and involve a lot of programming
- Professional programming is done in stages
- Beginning from the basic functionality, going up to the most advanced one
- Code is written in the form of small modules
- Sequences of inheritance starting from the most basic to the most advanced are slowly developed
- Benefits : Modular design, ease of maintenance, clarity of thought, faster error localization

Hierarchical Design

- In the real world, the systems to be developed are far too complex in nature, and involve a lot of programming
- Professional programming is done in stages
- Beginning from the basic functionality, going up to the most advanced one
- Code is written in the form of small modules
- Sequences of inheritance starting from the most basic to the most advanced are slowly developed
- Benefits : Modular design, ease of maintenance, clarity of thought, faster error localization

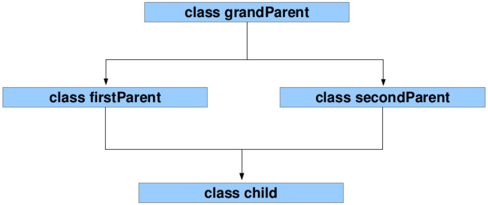
Hierarchical Design

- In the real world, the systems to be developed are far too complex in nature, and involve a lot of programming
- Professional programming is done in stages
- Beginning from the basic functionality, going up to the most advanced one
- Code is written in the form of small modules
- Sequences of inheritance starting from the most basic to the most advanced are slowly developed
- Benefits : Modular design, ease of maintenance, clarity of thought, faster error localization

Hierarchical Design

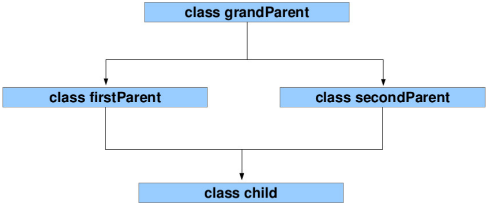
- In the real world, the systems to be developed are far too complex in nature, and involve a lot of programming
- Professional programming is done in stages
- Beginning from the basic functionality, going up to the most advanced one
- Code is written in the form of small modules
- Sequences of inheritance starting from the most basic to the most advanced are slowly developed
- Benefits : Modular design, ease of maintenance, clarity of thought, faster error localization

Multipath Inheritance



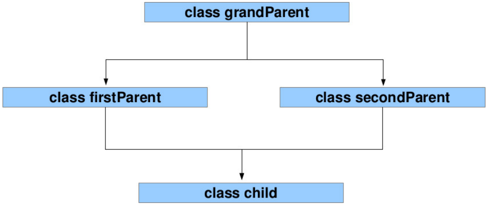
- firstParent and secondParent, both inherit from grandParent
- Child inherits from both the parents
- Child now has two copies of every inherited member of grandParent

Multipath Inheritance



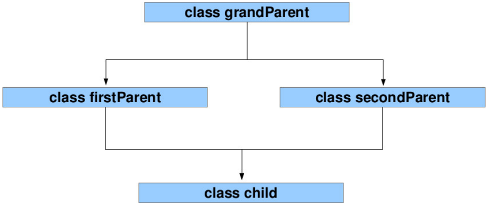
- firstParent and secondParent, both inherit from grandParent
- Child inherits from both the parents
- Child now has two copies of every inherited member of grandParent

Multipath Inheritance



- firstParent and secondParent, both inherit from grandParent
- Child inherits from both the parents
- Child now has two copies of every inherited member of grandParent

Multipath Inheritance



- firstParent and secondParent, both inherit from grandParent
- Child inherits from both the parents
- Child now has two copies of every inherited member of grandParent

A new keyword : “virtual”

- Problem: The child inherits the members of the grandparent twice
- This problem can be solved by declaring the inheritance of common base class as virtual

A new keyword : “virtual”

- Problem: The child inherits the members of the grandparent twice
- This problem can be solved by declaring the inheritance of common base class as virtual

A new keyword : “virtual”

```
class grandParent {
  ...
};
class Parent1 : virtual public grandParent {
  ...
};
class Parent2 : virtual public grandParent {
  ...
};
class child : public Parent1, public Parent2 {
  ...
};
```

- C++ checks that only one copy of the virtual base class is inherited.

A new keyword : “virtual”

```
class grandParent {
  ...
};
class Parent1 : virtual public grandParent {
  ...
};
class Parent2 : virtual public grandParent {
  ...
};
class child : public Parent1, public Parent2 {
  ...
};
```

- C++ checks that only one copy of the virtual base class is inherited.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Constructors and Inheritance

- Constructors : initialize new objects of a class
- If there is no constructor in any base class, then derived class need not necessarily have a constructor of its own
- However if any of the base classes has a constructor, then its mandatory for the derived class to have a constructor as well
- Base class is a part of the derived class after inheritance.
- Hence the derived class constructor should pass arguments to the base class constructor
- Base class constructors executed first in order of appearance in the derived class defenition, followed by the derived class constructor

Constructors and Inheritance

- Constructors : initialize new objects of a class
- If there is no constructor in any base class, then derived class need not necessarily have a constructor of its own
- However if any of the base classes has a constructor, then its mandatory for the derived class to have a constructor as well
- Base class is a part of the derived class after inheritance.
- Hence the derived class constructor should pass arguments to the base class constructor
- Base class constructors executed first in order of appearance in the derived class defenition, followed by the derived class constructor

Constructors and Inheritance

- Constructors : initialize new objects of a class
- If there is no constructor in any base class, then derived class need not necessarily have a constructor of its own
- However if any of the base classes has a constructor, then its mandatory for the derived class to have a constructor as well
- Base class is a part of the derived class after inheritance.
- Hence the derived class constructor should pass arguments to the base class constructor
- Base class constructors executed first in order of appearance in the derived class defenition, followed by the derived class constructor

Constructors and Inheritance

- Constructors : initialize new objects of a class
- If there is no constructor in any base class, then derived class need not necessarily have a constructor of its own
- However if any of the base classes has a constructor, then its mandatory for the derived class to have a constructor as well
- Base class is a part of the derived class after inheritance.
- Hence the derived class constructor should pass arguments to the base class constructor
- Base class constructors executed first in order of appearance in the derived class defenition, followed by the derived class constructor

Constructors and Inheritance

- Constructors : initialize new objects of a class
- If there is no constructor in any base class, then derived class need not necessarily have a constructor of its own
- However if any of the base classes has a constructor, then its mandatory for the derived class to have a constructor as well
- Base class is a part of the derived class after inheritance.
- Hence the derived class constructor should pass arguments to the base class constructor
- Base class constructors executed first in order of appearance in the derived class defenition, followed by the derived class constructor

Constructors and Inheritance

- Constructors : initialize new objects of a class
- If there is no constructor in any base class, then derived class need not necessarily have a constructor of its own
- However if any of the base classes has a constructor, then its mandatory for the derived class to have a constructor as well
- Base class is a part of the derived class after inheritance.
- Hence the derived class constructor should pass arguments to the base class constructor
- Base class constructors executed first in order of appearance in the derived class defenition, followed by the derived class constructor

Defining a constructor

- Assume that the derived class is inherited from base classes 1 to N

```
DerivedClass (List1, List2, ..., ListN,
              ListDerived) :
BaseClass1(List1),

BaseClass2(List2),

BaseClass3(List3),

BaseClassN(ListN) {
  // body of constructor of the derived class
}
```

Defining a constructor

- Assume that the derived class is inherited from base classes 1 to N

```
DerivedClass (List1, List2, ..., ListN,
                    ListDerived) :
BaseClass1(List1),

BaseClass2(List2),

BaseClass3(List3),

BaseClassN(ListN) {
  // body of constructor of the derived class
}
```

Defining a constructor

- Constructors of the base classes are invoked in order of their appearance in the definition of the derived class

Defining a constructor

```
class A {
public:
  int temp1;
  A(int i);
};
class B {
public :
  float temp2;
  B(float j);
};
class C: public A, public B {
public:
  char temp3;
  C(int i, float j, char c): A(i), B(j) {
    temp3 = c;
  }
};
```

Constructor: Initialization lists

- C++ provides an extra way to initialize a class via initialization list

```
ConstructorName(Argument List)
           : initialization section {
  // rest of the constructor definition
}
```

- The initialization occurs in the order of declaration, irrespective of the order in the list provided

Constructor: Initialization lists

- C++ provides an extra way to initialize a class via initialization list

```
ConstructorName(Argument List)
              : initialization section {
  // rest of the constructor definition
}
```

- The initialization occurs in the order of declaration, irrespective of the order in the list provided

Constructor: Initialization lists

- C++ provides an extra way to initialize a class via initialization list

```
ConstructorName(Argument List)
           : initialization section {
  // rest of the constructor definition
}
```

- The initialization occurs in the order of declaration, irrespective of the order in the list provided

Constructor: Initialization lists

```
class XYZ {
  int a, b, c;
public:
  XYZ(int i, int j, int k)
          : a(i), b(i+j), c(i+j-k) {
    cout<<''constructor XYZ''<<endl;
  }
};
class ABC : public XYZ {
  int z;
public:
  ABC(int i, int j, int k, int l)
          : XYZ(i,j,k), z(l) {
    cout<<''constructor ABC'';
  }
};
```

Nesting of classes

- Till now we saw that inheritance is a mechanism to extend the properties of one class to a new class
- C++ provides a second mechanism for the same purpose, which is in some sense more intutive, but used less
- This approach has the view that a new object can be formed by the collection of many other objects
- For eg : A car is made up of an Engine, four wheels, Headlights etc
- The objects of "base class" are directly used as "members" of the new class

Nesting of classes

- Till now we saw that inheritance is a mechanism to extend the properties of one class to a new class
- C++ provides a second mechanism for the same purpose, which is in some sense more intutive, but used less
- This approach has the view that a new object can be formed by the collection of many other objects
- For eg : A car is made up of an Engine, four wheels, Headlights etc
- The objects of "base class" are directly used as "members" of the new class

Nesting of classes

- Till now we saw that inheritance is a mechanism to extend the properties of one class to a new class
- C++ provides a second mechanism for the same purpose, which is in some sense more intutive, but used less
- This approach has the view that a new object can be formed by the collection of many other objects
- For eg : A car is made up of an Engine, four wheels, Headlights etc
- The objects of "base class" are directly used as "members" of the new class

Nesting of classes

- Till now we saw that inheritance is a mechanism to extend the properties of one class to a new class
- C++ provides a second mechanism for the same purpose, which is in some sense more intutive, but used less
- This approach has the view that a new object can be formed by the collection of many other objects
- For eg : A car is made up of an Engine, four wheels, Headlights etc
- The objects of "base class" are directly used as "members" of the new class

Nesting of classes

- Till now we saw that inheritance is a mechanism to extend the properties of one class to a new class
- C++ provides a second mechanism for the same purpose, which is in some sense more intutive, but used less
- This approach has the view that a new object can be formed by the collection of many other objects
- For eg : A car is made up of an Engine, four wheels, Headlights etc
- The objects of “base class” are directly used as “members” of the new class

Nesting of classes

```
class A {
  // definition of A
};
class B {
  // definition of B
};
class C {
  A objA;
  B objB;
  // definition of class C
}
```

All objects of the class C will contain the objects of A and B

Nesting of classes

```
class A {
  // definition of A
};
class B {
  // definition of B
};
class C {
  A objA;
  B objB;
  // definition of class C
}
```

All objects of the class C will contain the objects of A and B

Nesting of classes

- This sort of declaration or relationship between classes is called "Nesting" or "Containership"
- Such objects are created in 2 stages:
  - First the member objects are created using their own consrtuctors
  - Then rest of the ordinary members of the class are created

Nesting of classes

- This sort of declaration or relationship between classes is called “Nesting” or “Containership”
- Such objects are created in 2 stages:
  - First the member objects are created using their own consrtuctors
  - Then rest of the ordinary members of the class are created

Nesting of classes

- This sort of declaration or relationship between classes is called “Nesting” or “Containership”
- Such objects are created in 2 stages:
  - First the member objects are created using their own consrtuctors
  - Then rest of the ordinary members of the class are created

Nesting of classes

- This sort of declaration or relationship between classes is called "Nesting" or "Containership"
- Such objects are created in 2 stages:
  - First the member objects are created using their own consrtuctors
  - Then rest of the ordinary members of the class are created

Nesting of classes: With Constructors

- Since the member objects have to be created earlier, their constructors must be called earlier as well
- This is accomplished by the initialization list method for constructors

```
C(int x, float y, char z)
            : obj1(x), obj2(x,y,z) {
  // definition of the constructor
}
```

- The arguments in the initialization lists is for the member objects and not the respective classes
- The arguments passed to the member objects may or may not be from the argument list input to the constructor
- The constructors of the member objects are called in the order of their declaration in the class definition and not as per the order in the list

Nesting of classes: With Constructors

- Since the member objects have to be created earlier, their constructors must be called earlier as well
- This is accomplished by the initialization list method for constructors

```
C(int x, float y, char z)
           : obj1(x), obj2(x,y,z) {
  // definition of the constructor
}
```

- The arguments in the initialization lists is for the member objects and not the respective classes
- The arguments passed to the member objects may or may not be from the argument list input to the constructor
- The constructors of the member objects are called in the order of their declaration in the class definition and not as per the order in the list

Nesting of classes: With Constructors

- Since the member objects have to be created earlier, their constructors must be called earlier as well
- This is accomplished by the initialization list method for constructors

```
C(int x, float y, char z)
            : obj1(x), obj2(x,y,z) {
  // definition of the constructor
}
```

- The arguments in the initialization lists is for the member objects and not the respective classes
- The arguments passed to the member objects may or may not be from the argument list input to the constructor
- The constructors of the member objects are called in the order of their declaration in the class definition and not as per the order in the list

Nesting of classes: With Constructors

- Since the member objects have to be created earlier, their constructors must be called earlier as well
- This is accomplished by the initialization list method for constructors

```
C(int x, float y, char z)
            : obj1(x), obj2(x,y,z) {
  // definition of the constructor
}
```

- The arguments in the initialization lists is for the member objects and not the respective classes
- The arguments passed to the member objects may or may not be from the argument list input to the constructor
- The constructors of the member objects are called in the order of their declaration in the class definition and not as per the order in the list

Nesting of classes: With Constructors

- Since the member objects have to be created earlier, their constructors must be called earlier as well
- This is accomplished by the initialization list method for constructors

```
C(int x, float y, char z)
            : obj1(x), obj2(x,y,z) {
  // definition of the constructor
}
```

- The arguments in the initialization lists is for the member objects and not the respective classes
- The arguments passed to the member objects may or may not be from the argument list input to the constructor
- The constructors of the member objects are called in the order of their declaration in the class definition and not as per the order in the list

Nesting of classes: With Constructors

- Since the member objects have to be created earlier, their constructors must be called earlier as well
- This is accomplished by the initialization list method for constructors

```
C(int x, float y, char z)
              : obj1(x), obj2(x,y,z) {
  // definition of the constructor
}
```

- The arguments in the initialization lists is for the member objects and not the respective classes
- The arguments passed to the member objects may or may not be from the argument list input to the constructor
- The constructors of the member objects are called in the order of their declaration in the class definition and not as per the order in the list