4. Object Inheritance and Reusability

Inheritance

- In programming languages, inheritance means that the behavior and data associated with child classes are always an extension (that is, a larger set) of the properties associated with parent classes. A subclass will have all the properties of the parent class, and other properties as well. On the other hand, since a child class is a more specialized (or restricted) form of the parent class, it is also, in a certain sense, a contraction of the parent type.
- Inheritance is always *transitive*, so that a class can inherit features from super classes many levels away. That is, if class Dog is a subclass of class Mammal, and class Mammal is a subclass of class Animal, then Dog will inherit attributes both from Mammal and from Animal.

The Is-a Test

- There is a rule-of-thumb that is commonly used to test whether two concepts should be linked by an inheritance relationship. This heuristic is termed the is-a test. The is-a test says that to tell if concept A should be linked by inheritance to concept B, try forming the English sentence “A(n) A is a(n) B." If the sentence “sounds right" to your ear, then inheritance is most likely appropriate in this situation. For example, the following all seem like reasonable assertions:
  - *A Bird is an Animal*
  - *A Cat is a Mammal*
  - *An Apple Pie is a Pie*
  - *A TextWindow is a Window*
  - *A Ball is a GraphicalObject*
  - *An IntegerArray is an Array*
- On the other hand, the following assertions seem strange for one reason or another, and hence inheritance is likely not appropriate:
  - A Bird is a Mammal
  - An Apple Pie is an Apple
  - An Engine is a Car
  - A Ball is a Wall
  - An IntegerArray is an Integer

Reasons to Use Inheritance

- Inheritance as a means of code reuse. Because a child class can inherit behavior from a parent class, the code does not need to be rewritten for the child. This can greatly reduce the amount of code needed to develop a new idea.
- Inheritance as a means of concept reuse. This occurs when a child class overrides behavior defined in the parent. Although no code is shared between parent and child, the child and parent share the definition of the method.

Subclass, Subtype, and Substitution

- Consider the relationship of a data type associated with a parent class to a data type associated with a derived, or child, class in a statically typed object-oriented language. The following observations can be made:
  - Instances of the child class must possess all data members associated with the parent class.
  - Instances of the child class must implement, through inheritance at least (if not explicitly overridden) all functionality defined for the parent class. (They can also define new functionality, but that is unimportant for the present argument).
  - Thus, an instance of a child class can mimic the behavior of the parent class and should be indistinguishable from an instance of the parent class if substituted in a similar situation.
- The principle of substitution says that if we have two classes, A and B, such that class B is a subclass of class A, it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect.
- The term subtype is used to refer to a subclass relationship in which the principle of substitution is maintained, to distinguish such forms from the general subclass relationship, which may or may not satisfy this principle.
- In C++ only pointers and references truly support substitution; variables that are simply declared as value (and not as pointers) do not support substitution.

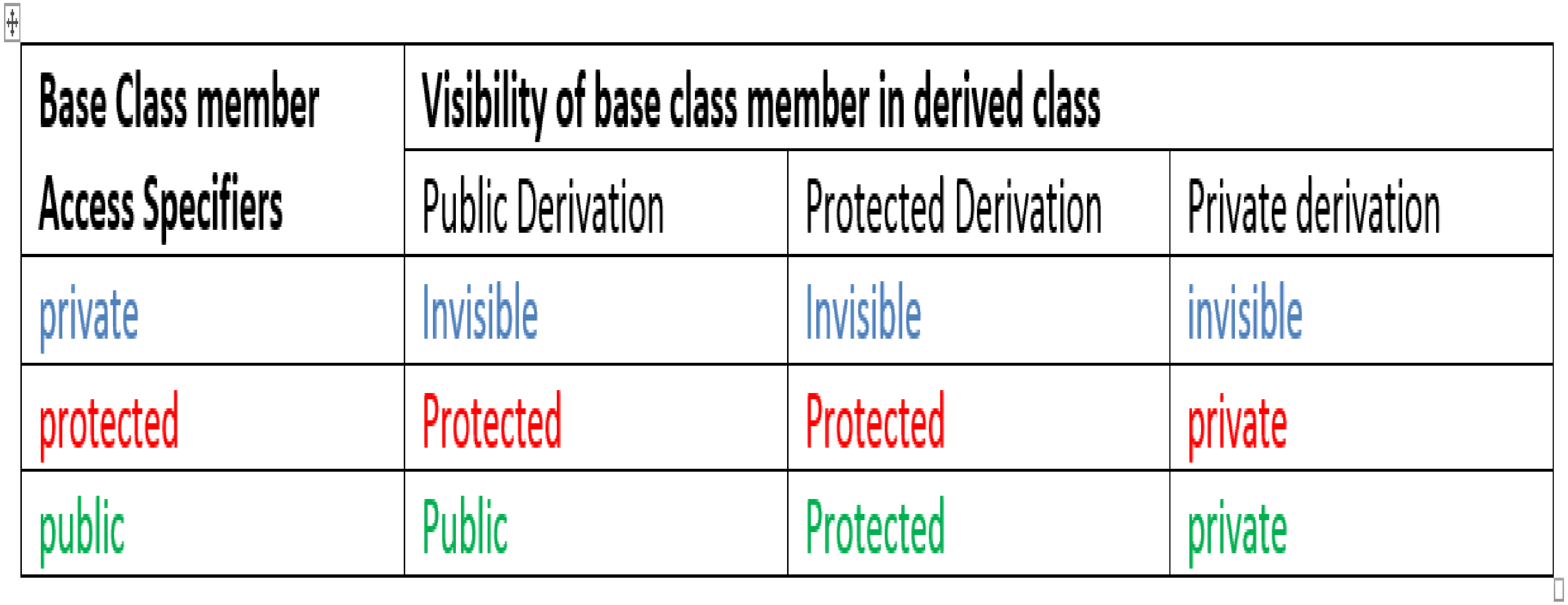
Public, Private and Protected

- A public feature is accessible to code outside the class definition, while a private feature is accessible only within the class definition. Inheritance introduces a third alternative.
- In C++ (also in C#, Delphi, Ruby and several other languages) a protected feature is accessible only within a class definition or within the definition of any child classes. Thus a protected feature is more accessible than a private one, and less accessible than a public feature.
- This is illustrated by the example

//program20

- The lines marked as error will generate compiler errors. The private feature can be used only within the parent class. The protected feature only within the parent and child class. Only public features can be used outside the class definitions.

Class Derivation



Base Class member Access Specifiers	Visibility of base class member in derived class		
	Public Derivation	Protected Derivation	Private derivation
private	Invisible	Invisible	invisible
protected	Protected	Protected	private
public	Public	Protected	private

Types of Inheritance

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

1. Single Inheritance

- When a class is derived from only one base class then such derivation is called single inheritance. In a single inheritance, the base class and derived class exhibits one to one relationship.
- General Form:

```
class derived_class : [access_to_base]base_class
{
      //...
};
```

- Example:

```
class A
{
      //...
};
class B : public A
{
      //...
};    //program23 and program24
```

Member Function Overriding Issue

- When defining a derived class, the data and function members can have the same name as that of the base class. The process of creating members in the derived class with the same name as that of the visible members of the base class is called overriding.
- It is called overriding because the new name overrides the old name inherited from base class. After overriding, when the members are accessed with the overridden name in the derived class, the derived class members are accessed.

2. Multiple Inheritance

- When class is derived from two or more base classes, then such type of inheritance is called multiple inheritance. In multiple inheritance, the derived class inherits some or all features of base classes from which it is derived.
- General Form:

```
class derived_class : [access_to_base] base_1, [access_to_base] base_2
{
        //…
};
```

Multiple Inheritance cont.

Example

```
class A
{
        //...
};
class B
{
        //...
};
class C : public A, public B
{
        //...
}; //program25 and program26
```

<u>Ambiguity Resolution in Multiple Inheritance</u>

When a function with same name appears in more than one base class then there exists an ambiguity about which function is used by the derived class during inheritance.

3. Multilevel Inheritance

- A new class that is derived from a base class can again serve as a base for further class derivation. The derivation of a class from another derived class is called multilevel inheritance.
- The visibility of base class member is same as in single inheritance. In case of second derivation, the members inherited as protected or public from the first inheritance can be visible from the derived class.
- For example:

```
class A
{
   //...
};
class B : public A
{
   //...
};
class C : public B
{
   //...
}; //program27
```

4. Hierarchical Inheritance

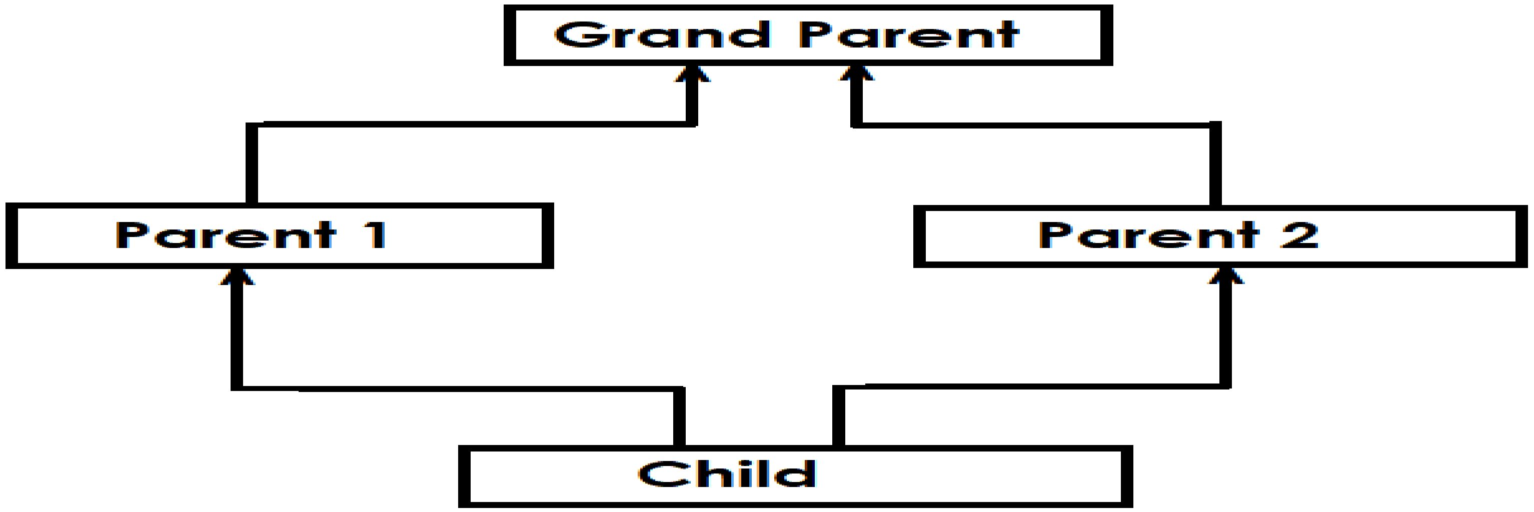
- When different classes are derived from a single base class, such type of inheritance is called as hierarchical inheritance.
- For example,

```
class A
{
   //...
};
class B : public A
{
   //...
};
class C : public A
{
   //...
}; //program28
```

5. Multipath Inheritance

- Consider a situation where all three kind of inheritances, namely multilevel, multiple and hierarchical inheritances, are involved.
- The Child has two direct base classes: Parent1 and Parent2 which themselves have a common base class Grandparent. The Child inherits the traits(properties) of Grandparent via two separate paths. The Grandparent is sometimes called as indirect base class.

//Program 29



Multipath Inheritance cont.

- All the public and protected members of Grandparent are inherited into Child twice, first via Parent1 and again via Parent2. This means Child would have duplicate sets of the members inherited from Grandparent. This introduce ambiguity and should be avoided.
- The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class while declaring the direct or intermediate base classes.
- When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and derived class.

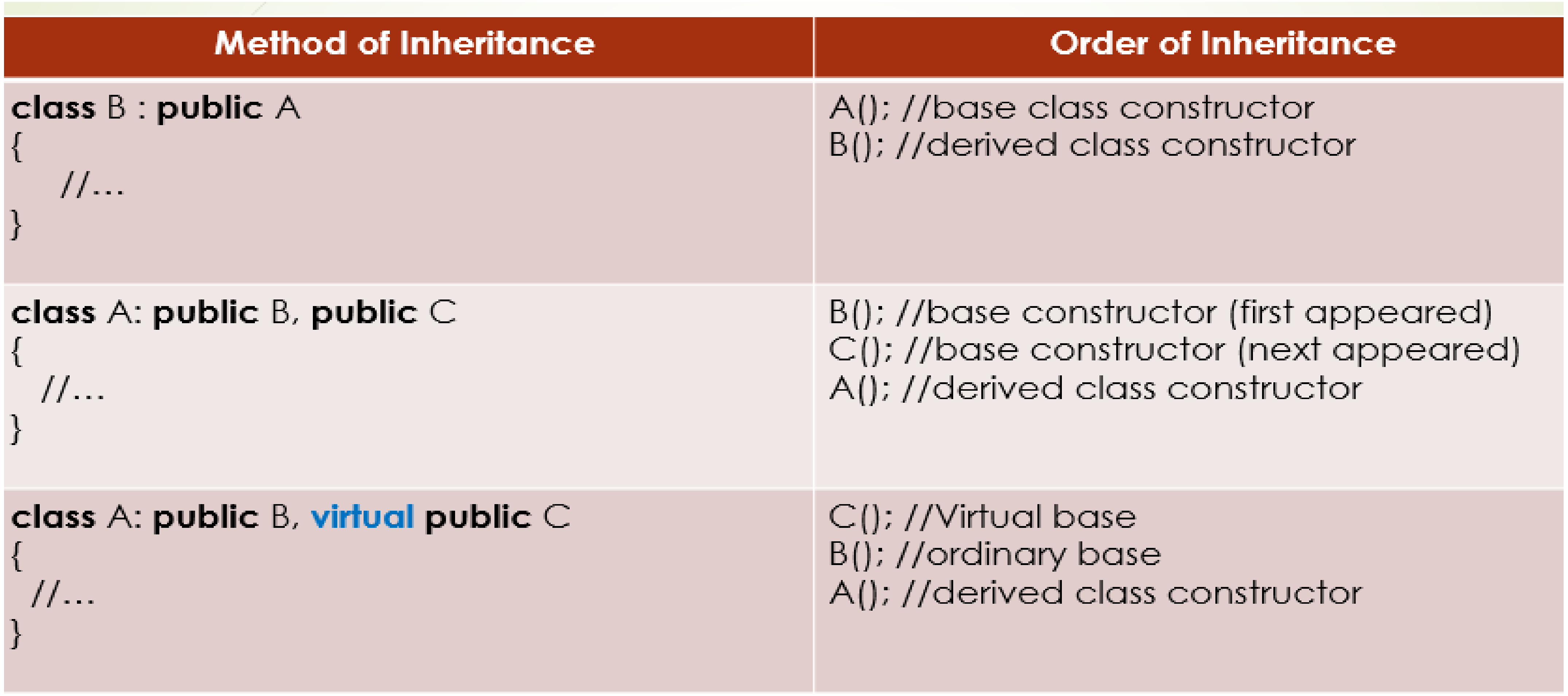
Multipath Inheritance: Example

```
class A
{
      //...
};
class B1 : virtual public A
{
      //...
};
class B2 : public virtual  B
{
      //...
};
class C : public B1, public B2
{
      //...
};
```

Constructors in Derived Classes

- If no base class constructor takes any argument, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. The derived class must pass arguments to the base class constructor since we usually create objects using the derived class.
- When both the derived class and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.
- In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructor will be executed in the order of inheritance.
- The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

Execution of Base Class Constructor



Method of Inheritance	Order of Inheritance

class B : **public** A
{
    //...
}
	A(); //base class constructor
	B(); //derived class constructor

class A: **public** B, **public** C
{
  //...
}
	B(); //base constructor (first appeared)
	C(); //base constructor (next appeared)
	A(); //derived class constructor

class A: **public** B, **virtual** **public** C
{
 //...
}
	C(); //Virtual base
	B(); //ordinary base
	A(); //derived class constructor

Forms of Inheritance

- **Specialization:** The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- **Specification**: The parent class defines behavior that is implemented in the child class but not in the parent class.
- **Construction:** The child class makes use of the behavior provided by the parent class but is not a subtype of the parent class.
- **Generalization:** The child class modifies or overrides some of the methods of the parent class.
- **Extension:** The child class adds new functionality to the parent class but does not change any inherited behavior.
- **Limitation**: The child class restricts the use of some of the behavior inherited from the parent class.
- **Variance:** The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
- **Combination:** The child class inherits features from more than one parent class. This is multiple inheritance.

1. Subclassing for Specialization (Subtyping)

- In subclassing for specialization, the new class is a specialized form of the parent class but satisfies the specifications of the parent in all relevant respects. Thus, in this form the principle of substitution is explicitly upheld.
- Here is an example of subclassing for specialization. A class Window provides general windowing operations (*moving, resizing, iconification, and so on*). A specialized subclass *TextEditWindow* inherits the window operations and in addition, provides facilities that allow the window to display textual material and the user to edit the text values.

2. Subclassing for Specification

- Another frequent use for inheritance is to guarantee that classes maintain a certain common interface--that is, they implement the same methods.
- The parent class can be a combination of implemented operations and operations that are deferred to the child classes. Often, there is no interface change of any sort between the parent class and the child class--the child merely implements behavior described, but not implemented, in the parent.
- This is in essence a special case of subclassing for specialization, except that the subclasses are not refinements of an existing type but rather realizations of an incomplete abstract specification. In such cases the parent class is sometimes known as an ***abstract specification class**.*
- The class *GraphicalObject* is an abstract class since it describes, but does not implement, the methods for drawing the object and responding to a hit by a ball. The subsequent classes *Ball*, *Wall*, and *Hole* then use subclassing for specification when they provide meanings for these methods.

3. Subclassing for Construction

- A class can often inherit almost all its desired functionality from a parent class perhaps changing only the names of the methods used to interface to the class or modifying the arguments in a certain fashion. This may be true even if the new class and the parent class fail to share the is-a relationship.
- The child class is not a more specialized form of the parent class, because we will never think of substituting an instance of the child class in a situation where an instance of the parent class is being used.

```
class Storable{
     void writeByte(unsigned char);}
};
class StoreMyStruct:public Storable{
     void writeStruct(MyStruct &aStruct);}
};
```

- Subclassing for construction tends to be frowned upon in *statically typed languages*, since it often directly breaks the principle of substitution (forming subclasses that are not subtypes); it is employed in *dynamically typed languages*. Many instances of subclassing for construction can be found in the *Smalltalk standard library.*

4. Subclassing for Generalization

- Using inheritance to subclass for generalization is the opposite of subclassing for specialization.
- Here, a subclass extends the behavior of the parent class to create a more general kind of object. Subclassing for generalization is often applicable when we build on a base of existing classes that we do not wish to modify Subclassing for generalization is often applicable when we build on a base of existing classes that we do not wish to modify, or cannot modify., or cannot modify.
- Consider a graphics display system in which a class *Window* has been defined for displaying on a simple black-and-white background. You could create a subtype *ColoredWindow* that lets the background color be something other than white by adding an additional field to store the color and overriding the inherited window display code that specifies the background be drawn in that color.
- Subclassing for generalization frequently occurs when the overall design is based primarily on data values and only secondarily on behavior. This is shown in the colored window example, since a colored window contains data fields that are not necessary in the simple window case.

5. Subclassing for Extension

- While subclassing for generalization modifies or expands on the existing functionality of an object, subclassing for extension adds totally new abilities.
- Subclassing for extension can be distinguished from subclassing for generalization in that the latter must override at least one method from the parent and the functionality is tied to that of the parent. Extension simply adds new methods to those of the parent, and the functionality is less strongly tied to the existing methods of the parent.
- An example of subclassing for extension is a *StringSet* class that inherits from a generic *Set* class but is specialized for holding string values. Such a class might provide additional methods for string-related operations--for example, “search by prefix," which returns a subset of all the elements of the set that begin with a certain string value. These operations are meaningful for the subclass but are not particularly relevant to the parent class.
- As the functionality of the parent remains available and untouched, subclassing for extension does not contravene the principle of substitution and so such subclasses are always subtypes.

6. Subclassing for Limitation

- Subclassing for limitation occurs when the behavior of the subclass is smaller or more restrictive than the behavior of the parent class. Like subclassing for generalization, subclassing for limitation occurs most frequently when a programmer is building on a base of existing classes that should not, or cannot, be modified.
- For example, an existing class library provides a double-ended-queue, or *deque*, data structure. Elements can be added or removed from either end of the deque, but the programmer wishes to write a stack class, enforcing the property that elements can be added or removed from only one end of the stack.
- In a manner like subclassing for construction, the programmer can make the Stack class a subclass of the existing Deque class and can modify or override the undesired methods so that they produce an error message if used. These methods override existing methods and eliminate their functionality, which characterizes subclassing for limitation.
- Because subclassing for limitation is an explicit contravention of the principle of substitution, and because it builds subclasses that are not subtypes, it should be avoided whenever possible.

7. Subclassing for Variance

- Subclassing for variance is employed when two or more classes have similar implementations but do not seem to possess any hierarchical relationships between the abstract concepts represented by the classes.
- The code necessary to control a mouse, for example, may be nearly identical to the code required to control a graphics tablet.
- Conceptually, however, there is no reason why class *Mouse* should be made a subclass of class *Tablet*, or the other way. One of the two classes is then arbitrarily selected to be the parent, with the common code being inherited by the other and device-specific code being overridden.
- Usually, however, a better alternative is to factor out the common code into an abstract class, say *PointingDevice*, and to have both classes inherit from this common ancestor. As with subclassing for generalization, this choice may not be available if you are building on a base of existing classes.

8. Subclassing for Combination

- A common situation is a subclass that represents a combination of features from two or more parent classes.
- A *teaching assistant*, for example, may have characteristics of both a *teacher* and a *student*, and can therefore logically behave as both. The ability of a class to inherit from two or more parent classes is known as multiple inheritance.

The Benefits of Inheritance

1. Software Reuse.
2. Code Sharing.
3. Consistency of Interface.
4. Software Components
5. Rapid Prototyping.
6. Information Hiding.
7. Polymorphism and Framework

Software Reusability

- When behavior is inherited from another class, the code that provides that behavior does not have to be rewritten.
- Many programmers spend much of their time rewriting code they have written many times before. With object-oriented techniques, these functions can be written once and reused.
- Other benefits of reusable code include increased reliability (the more situations in which code is used, the greater the opportunities for discovering errors) and the decreased maintenance cost because of sharing by all users of the code.

Code Sharing

- Code sharing can occur on several levels with object-oriented techniques. On one level, many users or projects can use the same classes.
- Another form of sharing occurs when two or more classes developed by a single programmer as part of a project inherit from a single parent class. When this happens, two or more types of objects will share the code that they inherit.
- This code needs to be written only once and will contribute only once to the size of the resulting program.

Consistency of Interface

- When two or more classes inherit from the same superclass, we are assured that the behavior they inherit will be the same in all cases.
- Thus, it is easier to guarantee that interfaces to similar objects are in fact similar, and that the user is not presented with a confusing collection of objects

Software Components

- Inheritance provides programmers with the ability to construct reusable software components. The goal is to permit the development of new and novel applications that nevertheless require little or no actual coding.
- Already, several such libraries are commercially available, and we can expect many more specialized systems to appear in time.

Rapid Prototyping

- When a software system is constructed largely out of reusable components, development time can be concentrated on understanding the new and unusual portion of the system.
- Thus, software systems can be generated more quickly and easily, leading to a style of programming known as rapid prototyping or exploratory programming.
- A prototype system is developed, users experiment with it, a second system is produced that is based on experience with the first, further experimentation takes place, and so on for several iterations.
- Such programming is particularly useful in situations where the goals and requirements of the system are only vaguely understood when the project begins.

Information Hiding

- A programmer who reuses a software component needs only to understand the nature of the component and its interface.
- It is not necessary for the programmer to have detailed information concerning matters such as the techniques used to implement the component.
- Thus, the interconnectedness between software systems is reduced. We earlier identified the interconnected nature of conventional software as being one of the principle causes of software complexity.

Polymorphism and Frameworks

- Software produced conventionally is generally written from the bottom up, although it may be designed from the top down.
- That is, the lower-level routines are written, and on top of these slightly higher abstractions are produced, and on top of these even more abstract elements are generated. This process is like building a wall, where every brick must be laid on top of an already laid brick.
- Normally, code portability decreases as one moves up the levels of abstraction.
- That is, the lowest-level routines may be used in several different projects, and perhaps even the next level of abstraction may be reused, but the higher-level routines are intimately tied to a particular application.
- Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be tailored to fit different applications by changes in their low-level parts.

Cost of Inheritance

- ***Execution speed***: Inherited methods, which must deal with arbitrary subclasses, are often slower than specialized code.
- ***Program size***: The use of any software library frequently imposes a size penalty.
- ***Message Passing Overhead***: Message passing is by nature a more costly operation than simple procedure invocation. As with overall execution speed, however, overconcern about the cost of message passing is frequently *penny-wise and pound-foolish*. For one thing, the increased cost is often marginal—perhaps two or three additional assembly-language instructions and a total time penalty of 10 percent.
- ***Program Complexity***: Although object-oriented programming is often touted as a solution to software complexity, in fact, overuse of inheritance can often simply replace one form of complexity with another. Understanding the control flow of a program that uses inheritance may require several multiple scans up and down the inheritance graph. (*yo-yo problem*)

Software Reuse

- Two different approaches to reuse:
  - Inheritance – the *is-a* relationship.
  - Composition – the *has-a* relationship.

Example: Building Sets from Lists

Suppose we have already a ***List*** data type with the following behavior. We want to build the ***Set*** data type (elements are unique).

```
class List{
    public :
        void add(int);
        int includes(int);
        void remove(int);
        int firstElement();
};
```

Using Inheritance

Only need specify what is *new* - the addition of methods; everything else is given for free.

```
class Set : public List{
     public:
          void add(int);
};
void Set::add(int x)
{
     if (!includes(x))//only include if not already there
          List::add(x);
}
```

Using Composition

Everything must be redefined, but implementation can make use of the list data structure.

```
class Set {
      private:
      List data;

      public :
      void add(int);
      int includes(int);
      void remove(int);
      int firstElement();
};
```

```
void Set::add(int x)
{
     if (!data.includes(x))
     data.add(x);
}
int Set::includes(int x)
{ return data.includes(x); }
void Set::remove(int x)
{ data.remove(x); }
int Set::firstElement()
{ return data.firstElement(); }
```

Composition vs Inheritance

- Not substitutable
- Simple, makes all methods explicit
- Easy to change underlying data structure
- Can only access public stuff of the data-structure
- Longer code, but everything specified locally

- Substitutable
- Allows for all, even inappropriate, parent class methods
- Trickier, especially if uses depend on parent methods
- Can access public and protected parent class stuff
- Shorter code, but must fully understand parent class, too

Is-a vs Has-a Relationship

- Only the Is-a relation is related to inheritance. The OO concept of the Is-a relationship is called specialization. Most OO languages use classes to implement the concept of types and inheritance as only implementation of specialization. In an Is-a relationship A Is-a B if A is a specialization of B. This means that every instance/object of type A can do at least the things that B can do, i.e. it shows the same behavior for the same message (messages are usually implemented as methods). In the case of C++ the previous example for inheritance looks like this:

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

- The Has-a relationship is not called inheritance but composition. As the name suggests A has a member variable of type B. A is not a specialization of B and thus does not inherit anything from B. The corresponding example in C++ would look like this:

Is-a vs Has-a Relationship cont.

```
class B {
};
class A {
  B b;
};
```

- As you can see class A does not have to show the same behavior as B. This means that A does not have to respond to the same messages as B (speaking of the concept), i.e. A does not have to have the same methods as B (speaking of the implementation in C++). They can be completely different or partially overlapping.
- In the first example where A Is-a B this means that everywhere where your program expects an object of type B you can also provide an object of type A as it is also an object of type B. In the second example this is not the case. However, you cannot use an object of type B where an object of type A is expected.