Lecture 8: Advanced C++ Topics

Read: Chpt. 8, Carrano

Introduction to Inheritance:

Sometimes we wish to define a data type that is a ***specialization*** or ***extension*** of another data type.

Example: Consider the class ***animal***:
Dog is a special type of animal.
Fish is also an animal.
Collie is a special type of dog (and also an animal).
There is a fish called Wanda!

$$\begin{tikzcd} & \text{Animal} \arrow[dl] \arrow[dr] & \\ \text{Dog} \arrow[d] & & \text{Fish} \arrow[d] \\ \text{Collie} & & \text{Wanda} \end{tikzcd}$$

Q: How does inheritance function in an OO language?

In OO languages like C++, we use inheritance to create a new ***(derived)*** class from an existing ***(base)*** class.

Inheritance and Class Relation:

Inheritance allows a class to derive properties from an existing class.

$$\begin{tikzcd} \text{Base class (superclass)} \arrow[d] \\ \text{Derived class (subclass)} \end{tikzcd}$$

A derived class will inherit all the member data and member functions, *except constructors and destructor*, from its base class.

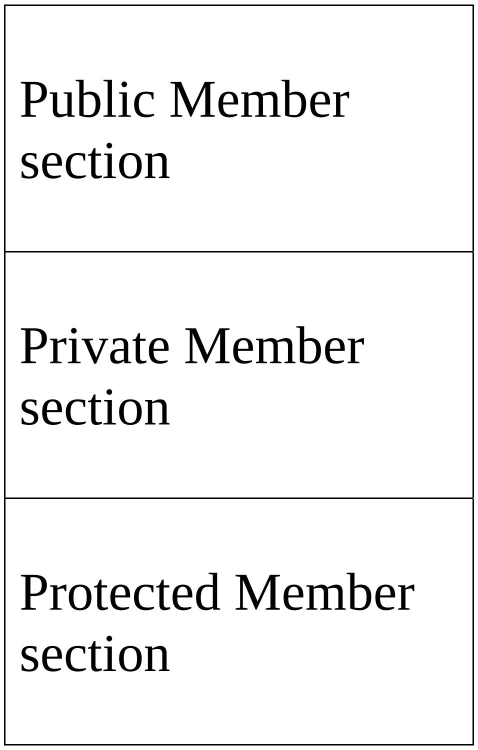
Syntax:

class derivedClassName: accessModifier baseClassName

Access Modifier	**Type of Inheritance**
public	public
private	private
protected	protected

Class Structure & Access Modifiers:

Class Structure:



Properties:

- Public members can be used by anyone.
- Private members can only be used by member functions (and *friends*) of the class.
- Protected members can only be used by member functions (and *friends*) of the class and its derived class.

Inheriting Data/Function Members:

	Public In.	**Protected In.**	**Private In.**
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Private	Private	Private

Public Inheritance:

Public/protected members of base class

$\rightarrow$ Public/protected members of derived class

Protected Inheritance:

Public/protected members of base class

$\rightarrow$ Protected members of derived class

Private Inheritance:

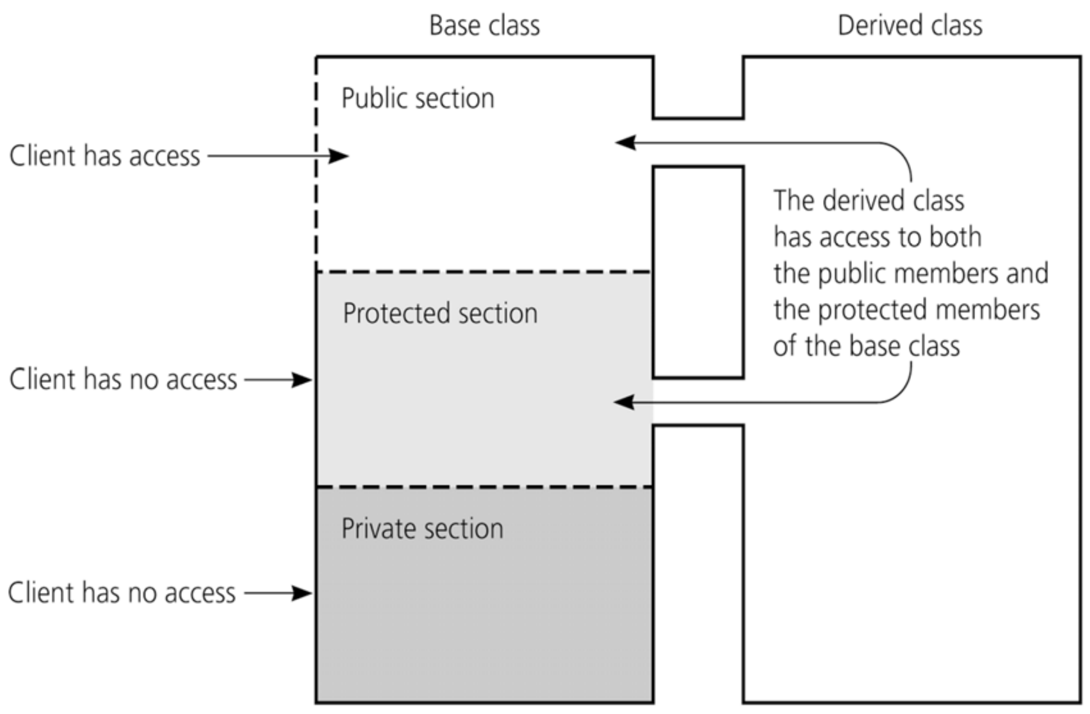
Public/protected members of base class

$\rightarrow$ Private members of derived class

Inheritance and Access Modifiers:

- The ***public*** members of a base class are accessible to any function.
- The ***protected*** members of a base class are private to any function except those member functions of a class that is derived directly, or indirectly, by an inheritance chain from the base class, regardless of length.
- The ***private*** members of a base class are private to any function; direct use of these members in a derived class would be illegal.

Accessibility & Inheritance:

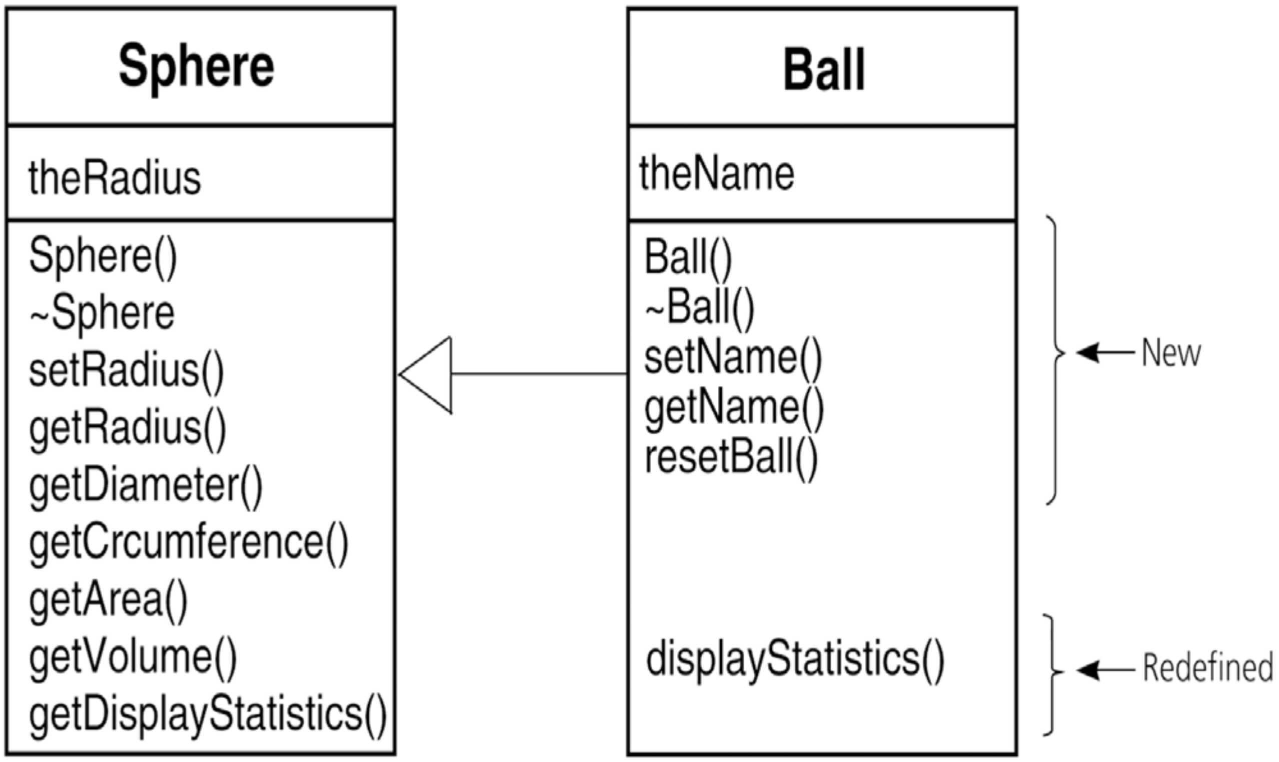


Inheritance and Member Data/Functions:

- The derived class can define additional member variables, functions, or both. If class D is derived from class B, then class D will have all the features of class B plus some addition added features, if desired, as well.
- When a derived class is defined, the prototypes of *inherited* member functions are not included in the derived class. However, all prototypes of all *new* member functions, including redefined and overloaded functions, must be given.
- If a derived class requires a different implementation for an inherited member function, then the function may be ***redefined*** in the derived class. When a member function is redefined, you must list its prototype in the definition of the derived class, even though the prototype is the same as in the base class.

Warning: Do not confuse function ***redefinition*** with function ***overloading***. When you redefine a function, the new function given in the derived class has the exact same number and types of parameters as the function in the base class. By contrast, suppose that in the derived class, there is a (new) function with the same name, but a different number of parameters or a different sequence of parameter types (or both). Then the derived class will have both functions. This is an example of overloading.

Example: Consider the following Sphere-Ball classes.



```
class Sphere
{
public:
// constructors:
   Sphere();
   Sphere(double initialRadius);
   // copy constructor and destructor supplied
   // by the compiler

// Sphere operations:
   void setRadius(double newRadius);
   double getRadius() const;
   double getDiameter() const;
   double getCircumference() const;
   double getArea() const;
   double getVolume() const;
   void displayStatistics() const;

private:
   double theRadius; // the sphere's radius
};  // end class
```

```
class Ball: public Sphere       // public inheritance
{
public:
// constructors:
   Ball();
   Ball(double initialRadius,const string& initialName);
   // Creates a ball with radius initialRadius and
   // name initialName.
   // copy constructor and destructor supplied
   // by the compiler

// additional or revised operations:
   void getName(string& currentName) const;
   // Determines the name of a ball.

   void setName(const string& newName);
   // Sets (alters) the name of an existing ball.

   void resetBall(double newRadius, const string& newName);
   // Sets (alters) the radius and name of an existing
   // ball to newRadius and newName, respectively.

   void displayStatistics() const;       // Function redefinition
   // Displays the statistics of a ball.

private:
   string theName; // the ball's name
}; // end class
```

```
Ball::Ball() : Sphere()
{
   setName("");
}  // end default constructor

Ball::Ball(double initialRadius,const string& initialName)
          : Sphere(initialRadius)
{
   setName(initialName);
}  // end constructor

void Ball::getName(string& currentName) const
{
   currentName = theName;
}  // end getName

void Ball::setName(const string& newName)
{
   theName = newName;
}  // end setName

void Ball::resetBall(double newRadius,
                     const string& newName)
{
   setRadius(newRadius);
   setName(newName);
}  // end resetBall
```

```
void Ball::displayStatistics() const
{
   cout << "Statistics for a " << theName << ":";
   Sphere::displayStatistics();
} // end displayStatistics
```

Remarks:

- An instance of Ball has two data members, theName and the inherited theRadius.
- The data member theRadius can not be accessed directly by an instance of Ball; it can only be accessed through the use of Sphere’s public member functions setRadius and getRadius.
- An instance of Ball has all the member functions defined in Sphere; a new constructor; a new compiler generated copy constructor and destructor; additional Ball functions getName, setName, and resetBall; and a newly redefined displayStatistics.
- The constructors for Ball can invoke the corresponding constructors in Sphere by using an initializer syntax.
- The redefined displayStatistics in Ball can call the inherited version of displayStatistics in Sphere using the scope resolution operator.

Another Example on Inheritance:

```
class Student
{
    public:
      Student();
      Student(const Student& s);
      Student(string name, int ID);

      int       getID() const;
      string    getName() const;
      int       getNumHoursTaken() const;
      int       getNumGradePointsEarned() const;
      double    getGPA() const;
      void      print(ostream& os) const;
      void      updateGrades(int moreHours,int morePoints);

    private:
      string    mName;
      int       mID;
      int       mHoursTaken;
      int       mGradePointsEarned;
};
```

Given the above Student class, suppose that we need to add a graduate program. In addition to all of the above information, which continues to be applicable to graduate students, we also need to include the following information:

- Support type (TA, RA, Fellowship, None).
- Advisor name.
- Thesis topic.

Q: How can we implement this new GraduateStudent class?

Traditional (Non-OO) Approach:

- Modify Student definition to include newGraduateStudent information.
- Add extra data member, methods, and modify existing methods (e.g., *print*) if necessary.
- Recompile this new class.

Q: What's wrong with this approach?

Problems with Traditional (Non-OO) Approach:

1. All students have these extra fields and methods.
2. We had to modify our earlier perfectly good, working code to add things unrelated to the majority of our students.
3. We have to re-examine and possibly modify all existing code we wrote using the Student class interface, and we gain nothing from the effort!

Q: Any better approach?

Modern Object Oriented Approach:

1. Using the concept of ***Inheritance***, we can derive a new type which is a specialized type of Student.
2. We only need to add new code, not modify existing code.
3. None of our earlier code working on regular Student objects needs to be changed at all.

Q: How do we extend the definition of the Student class to the new GraduateStudent class?

Using public inheritance!

Example of Public Inheritance:

```
class GraduateStudent : public Student     // public inheritance
{
    public:
      enum SupportType {TA, RA, None};

      GraduateStudent();
      GraduateStudent(const GraduateStudent& s);
      GraduateStudent(string name, int ID);
      GraduateStudent(string name, int ID,
              SupportType sType, string advisor, string topic);

      SupportType    getSupportType() const;
      string         getThesisAdvisor() const;
      string         getThesisTopic() const;
      void           print(ostream& os) const;

    private:                            // new data fields
      SupportType    mSupportType;
      string         mThesisAdvisor;
      string         mThesisTopic;
};
```

Recall that, except constructors and destructor, instances of “GraduateStudent” have all instance variables and methods of “Student” plus the additional ones.

Memory Layout for an Instance of “Student”

mName
mID
mHoursTaken
mGradePointsEarned

Memory Layout for an Instance of “GraduateStudent”

mName
mID
mHoursTaken
mGradePointsEarned
mSupportType
mThesisAdvisor
mThesisTopic

In file: client.c++

```
void workWithStudents()
{
   Student s1(“Sally Smith”, 143298, 0, 0);
   GraduateStudent g1(“Hector”, 110091, 0, 0, None, “”, “”);
   s1.print(cout);          // Using print in Student
   g1.print(cout);          // Using print in GraduateStudent
}
```

s1:

“Sally Smith”
143298
0
0

g1:

“Hector”
110091
0
0
None
“”
“”

Using “GraduateStudent” and “Student” Methods:

```
void report(GraduateStudent g)
{
   if (g.getSupportType() != GraduateStudent::None)
        cout  <<  g.getThesisAdvisor()
                <<  “has a student working on ”
                <<  g.getThesisTopic() << “.\n”;
        cout  <<  g.getName() <<  “ has GPA = ”
                <<  g.getGPA()  <<  endl;
}
```

Observe that public inheritance reflects “**is-a**” relationship. Since a “GraduateStudent” is a “Student”, g.getName() and g.getGPA() are both meaningful!

However, not all students are graduate students!

Consider the following segment of code:

```
   Student  s;
   ...
   cout    << “The advisor for ”
             << s.getname()
             <<  “ is ”
             <<  s.getThesisAdvisor()      // Illegal!
             << endl;
      …
```

Here, “s.getname()” is legal since getname() is defined for any Student. However, “s.getThesisAdvisor()” is illegal since ThesisAdvisor is only assigned to GraduateStudent! In C++ terms, the getThesisAdvisor() method is only defined for instances of GraduateStudent!

An Incorrect GraduateStudent Method Implementation:
Consider the following implementation of the GraduateStudent print method.

```
void GraduateStudent::print(ostream& os)
{
    os  << “Student: ” << mName << ‘(‘
        << mID << “) has taken ”
        << mHoursTaken << “ hours and earned ”
        << mGradePointsEarned << “ grade points.\n”;
    os  << “Advisor: ”
        << mThesisAdvisor << ‘\n’;
}
```

Q: What’s wrong?

Remedy:
(1) Change “private” to “protected” in class Student.
(2) Use public accessor methods.

```
void GraduateStudent::print(ostream& os)
{
   os  << “Student: ” << getName() << ‘(‘
       << getID() << “) has taken ”
       << getNumHoursTaken() << “ hours and earned ”
       << getNumGradePointsEarned() << “ grade points.\n”;
   os  << “Advisor: ”
       << mThesisAdvisor << ‘\n’;
}
```

Since the Student class already has a print method, the best way to implement the GraduateStudent print method is to use the inherited print method.

```
 void GraduateStudent::print(ostream& os)
 {
   Student::print(os);
   os  << “Advisor: ”
       << mThesisAdvisor  << ‘\n’;
 }
```

Example of Private Inheritance:

```
class blaster
{
    public:
        ...
        void  blastOff();

    private:
        ...
};


class lifter: private blaster
{
        ...
};
void someFunction(lifter myLifter)
{
   myLifter.blastOff();              // Illegal!
          …
 }
```

Warning: This is illegal! When using private inheritance, clients of class “lifter” cannot use an instance of “lifter” to access *any* method or instance variable (whether public or private!) of “blaster”:

Q: Why private inheritance?

Why would we ever want such a relationship between classes?

Private inheritance can be used when we want to implement one ADT using another ADT. An instance of the derived class is not a special case of the base class. Instead, private inheritance reflects “as-a” (“is-implemented-using”; “is-implemented-in-terms-of”) relationship.

Consider the implementation of Stack class using List class.

```
class List
{
    …
    // includes the public method: insert(...);
};

class Stack : private List
{
    …
    // includes the public method: push(...);
};
```

Observations:

1. A Stack is not a special type of List.
2. The private derivation prevents clients from knowing (or at least exploiting) the fact that our implementation uses a linked list:

```
Stack  stk;
    ...
stk.insert(...);       // Illegal!
stk.push(...);         // OK…
```

3. The *implementation* of “push” can invoke the List “insert” method.
4. The idea is that the *clients* of Stack can only use Stack operations since to do otherwise will violate the principal of Abstraction.
5. Private inheritance is hard to justify. It is usually much better to use a “has-a” relationship when using one ADT to implement another.

Example of Private Inheritance:

```
class blaster
{
public:
        ...
        void  blastOff();

private:
        ...
};

class lifter: protected blaster
{
        ...
};

void someFunction(lifter myLifter)
{
      myLifter.blastOff();
             …
 }
```

Three Primary Class Relationships:

Relationship	**Means**	**In C++**
is-a	“Is a special kind of”	Public inheritance
as-a	“Implemented in terms of”	Private inheritance
has-a	“has an instance of”	An object has an instance variable of some other class

Visibility and Accessibility Rules:

When studying accessibility of methods and instance variables of a class, *B*, we have focused on two major accessibility categories, public and private, and two major execution contexts, within implementations of methods of class *B* and within “client” code, including within implementations of methods of unrelated classes.

With inheritance, a third accessibility category, protected, and a third context, within implementations of methods of a subclass *D* of *B*, must be considered.

$$\begin{tikzcd}
\text{Base class B} \arrow[d] \\
\text{Derived class D}
\end{tikzcd}$$

Visibility vs. Accessibility:

An identifier is visible in a given context if the definition of the identifier is in the scope of the given context. It is accessible in a given context if (1) it is visible, and (2) permission has been granted for the identifier to be used in the given context

Visibility and Accessibility Relationships

For instance variables and methods declared in base class B in section:	V/A in client code via instances of				V/A in method implementations of		
	Base B	A derived class D where the derivation path from B to D is all public	A derived class D where at least one derivation on the path from B to D is protected, but none are private	A derived class D where at least one derivation on the path from B to D is private	B (i.e., base class methods)	D (when there is no private inheritance along the path from B to *D's parent*)	D (when there is at least one private inheritance along the path from B to *D's parent*)
public	A	A	V	V	A	A	V
protected	V	V	V	V	A	A	V
private	V	V	V	V	A	V	V

"V" indicates visible, but not accessible
"A" indicates accessible (hence also visible)

Virtual Functions & Inheritance:

Sometimes subclasses need to *modify* or *totally replace* the action implemented by a particular method in a base class. We say that the subclass **overrides** the inherited method.

To make this possible:

- The method must be defined as “virtual” in the base class. It can then be redefined in each derived class.
- A derived class does not have to redefine (override) a virtual function.
- In the base class, you must begin the function declaration with the keyword virtual.
- The derived function must have exactly the same prototype as the inherited function in the base class.
- The keyword virtual does not need to be used in the derived functions.
- Friend functions cannot be virtual, nor can **constructor** functions, although **destructor** functions can be virtual.

Example:

```
class Animal
{
public:
    ...
    virtual void breathe( );   // uses the animal's nose
    ...
};

class Fish: public Animal
{
public:
    ...
    virtual void breathe( );  // uses the fish's gills
    ...
}
```

Remarks:

- By declaring "breathe( )" to be virtual, the designer of "Animal" has granted permission to designers of subclasses of "Animal" to **override** "breathe( )".
- Being "virtual" in "Fish" means only that subclasses of "Fish" can override the version of "breathe( )" they inherited from "Fish", should they choose to do so.

Another Example:

```
class baseClass
{   public
        virtual void print()
        {
            cout << “print from baseClass” << endl;
        }
}   // end baseClass

class firstClass : public baseClass
{   public
        void print()
        {
            cout << “print from firstClass” << endl;
        }
}   // end firstClass

class secondClass : public baseClass
{
}   // end secondClass
```

```
class thirdClass : public baseClass
{   public
        void print()
        {
            cout << “print from thirdClass” << endl;
        }
}   // end thirdClass

class standardClass : public thirdClass
{
}   // end standardClass
```

Class hierarchy:

$$\begin{tikzcd}
 & \text{baseClass} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{firstClass} & \text{secondClass} & \text{thirdClass} \arrow[d] \\
 & & \text{standardClass}
\end{tikzcd}$$

Consider the following set of function calls:

```
void printout ()
{
    baseClass         base;
    base.print();
    firstClass        first;
    first.print();
    secondClass       second;
    second.print();
    thirdClass        third;
    third.print();
    standardClass     standard;
    standard.print();
}   // end printout
```

Output:

```
  print from baseClass
  print from firstClass
  print from baseClass
  print from thirdClass
  print from thirdClass
```

Another Example:

```
class base
{    public:
          base();
          base(base& b);
          base(int val);
          ...
          virtual   void      doSomething();
          virtual   void      doSomethingElse();
          void                makeNoise();
     private:
          int                 bValue;
};
class derived: public base
{    public:
          derived();
          derived(derived& d);
          derived(int baseVal, int derivedVal);
          …
          virtual   void   doSomething();
                    void   doSomethingElse();
                    void   makeNoise();
     private:
          int       dValue;
};
```

Consider the following set of method calls:

```
void clientFunction()
{     base       b;
      derived  d;

      b.doSomething();    // base's doSomething
      d.doSomething();     // derived's doSomething

      base* bp = new derived;
      bp->doSomething();   // derived's doSomething

      base* bp = new base;
      bp->doSomething();   // base's doSomething

      derived* dp = new derived;
      dp->doSomething();   // derived's doSomething

      derived* dp = new base;
      dp->doSomething();   // illegal
      }
```

Q: How does the compiler decide what method implementation to invoke?

Virtual Functions and Dynamic (late) Binding:
Static vs. Dynamic Types:

The *static type* of an identifier is the type used to declare the identifier. The *dynamic type* of an identifier is the type of the object associated with the identifier at a given point in time during the execution of the program.

Dfn: Any class that contains a virtual function is called a ***polymorphic*** class.

Every polymorphic class has a virtual method table (VMT)

- Contains pointers to the versions of the virtual functions that are appropriate for the calling object.
- Enables late binding.

Pure Virtual Function & Abstract Class:

In some cases, we may not want to define the virtual function in the base class at all. Instead, the base function should simply provide a kind of placeholder for the virtual function and leave it up to the derived classes to specify the individual methods. A pure virtual function is a virtual function that is declared but not defined in a base class.

Syntax:

```
virtual type function_name(parameter_list)=0;
```

Remarks:

1. When a base class contains a pure virtual function, it is called an abstract class.
2. An abstract class is a general class that lays the foundation for its subclasses that define their own methods.
3. Each subclass of an abstract class must define the abstract function(s).
4. Since there is no definition for at least one virtual function in an abstract class, you can not create an object of that class.

Example of an Abstract Class:

```
class EquidistantShape
{
public:
    void setRadius (double newRadius);
    double getRadius() const;
    virtual void displayStatistics() const = 0;
…
};
```

Reasons for Overriding Functions:

1. Replace functionality:

```
void base::doSomething()
{
    bValue += 100;
}

void derived::doSomething()
{
    dValue += 100;
}
```

2. Augment functionality:

```
void base::doSomethingElse()
{
      ... // whatever base operations are appropriate
}

void derived::doSomethingElse()
{     ... // some initial stuff, possibly nothing
      base::doSomethingElse();
      ... // some other stuff, possibly nothing
}
```

Remark: From a language/syntactic point of view, there is no difference between “replacing” and “augmenting” the functionality of a method. It is strictly a matter of how you choose to implement overridden methods.

Friends:

- Functions and classes can be friends of a class.
- Friend functions can access the private and protected members of the class.
- Member functions of a friend class have access to the private and protected parts of the class to which it is a friend.
- Friend functions are not members of the class.
- A friend of a base class is not a friend of a derived class.

Example:

A List class can be a friend of the ListNode class:

```
class ListNode
{
private:
  …
     friend class List;
};
```