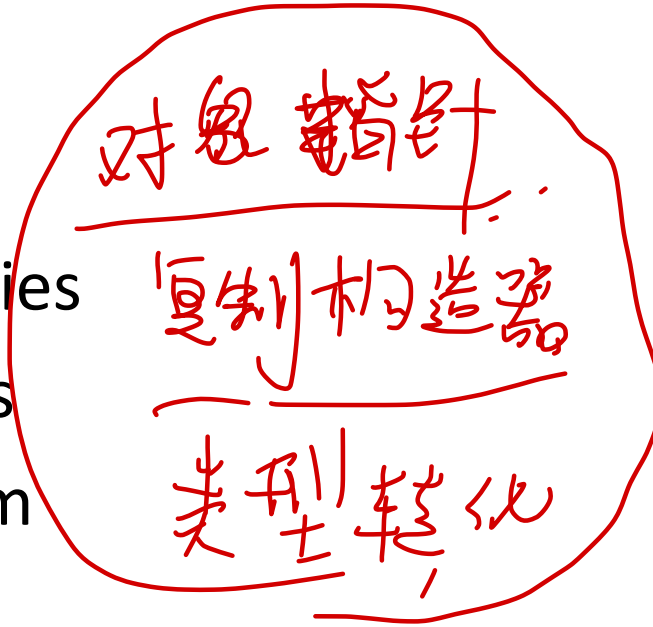
Chapter 12: Adding Functionality to Your Classes

C++ FOR ENGINEERS AND SCIENTISTS

Objectives

In this chapter you'll learn about:

- Providing class I/O capabilities
- Providing class conversion capabilities
- Class scope and duration categories
- Class inheritance and polymorphism
- Virtual functions
- Common programming errors



Providing Class I/O Capabilities

- In this section, you will be introduced to:
  - How to adapt the `cout` and `cin` classes to work with user-defined data types
    - The `Complex` class is used as an example
  - How to overload the insertion and extraction operators by creating operator functions
  - How to provide these functions with access to private member data

Step 1: Overload the `ostream` Insertion Operator <<

- The insertion operator is being overridden in the existing C++ class `ostream`, not one in the user-defined `Complex` class
- This is possible only because the insertion operator function is defined as public in the `ostream` class

Step 1: Overload the `ostream` Insertion Operator << (continued)

- The operator definition must adhere to the following syntax required by the `ostream` class:

```
ostream& operator<<(ostream& variableName, const className& objectName)
{
  // statements placing data on the output stream
  // referenced by the identifier variableName

  return variableName;
}
```

Step 1: Overload the `ostream` Insertion Operator << (continued)

- For overloading the insertion operator to display `Complex` objects, the user-selected names `out` and `num` and the class name `Complex` are used:

```
ostream& operator<<(ostream& out, const Complex& num)
{
  // statements placing data on the output stream
  // referenced by the identifier out

  return out;
}
```

Step 2: Provide Access to the `Complex` Class

- Access to the `Complex` class is achieved by declaring the overloaded function as a friend
- Include the overloaded function's prototype in the `Complex` class's declaration section, preceded by the keyword `friend`

```
friend ostream& operator<<(ostream&, const Complex&);
```

Refer to page 686 for more explanations and examples

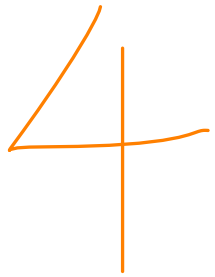
Adapting the `istream` Object `cin`

- The same two steps used for `cout` apply to `cin`
  - Step 1: Overload the `istream` operator, >>, with an overloaded operator function
  - Step 2: Provide access to the `Complex` class by making this function a friend of the `Complex` class
- Program 12.2 includes the function prototypes for the overloaded insertion and extraction operator functions and their definitions

Refer to page 690 for more explanations and examples

Providing Class Conversion Capabilities

- Possibilities for conversion between data types include:
  - Conversion from built-in type to built-in type
  - Conversion from class type to built-in type
  - Conversion from built-in type to class type
  - Conversion from class type to class type
- A conversion makes sense only when there is a meaningful relationship between data types



Built-in to Built-in Conversion

- Can be implicit or explicit
- Implicit conversion occurs in C++'s operations
- Explicit conversion occurs when a cast is used
- Two cast notations:
  - C notation: *(dataType)expression*
  - C++ notation: *dataType(expression)*

Refer to page 694 for more explanations and examples

Class to Built-in Conversion

- The conversion operator function
  - Converts from user-defined data type to built-in data type
  - Is a member function having the same name as the built-in data type or class
  - When name is the same as built-in type, used to convert from class to built-in data type
    - Conversion operator for class to long conversion would be named operator `long()`
  - Has no explicit argument or return type

Refer to page 694 for more explanations and examples

Built-in to Class Conversion

- User-defined casts for converting a built-in type to a class type are created by using constructor functions
- Type conversion constructor:
  - A constructor whose first argument is not a member of its class and whose remaining arguments, if any, have default values
  - If the first argument is a built-in type, then constructor can be used to cast the built-in to the class type

Built-in to Class Conversion (continued)

- Constructor function
  - Is called implicitly to initialize an object
  - Can be called explicitly after all objects have been declared
- Sample conversion construction:

```
// constructor for converting from long to Date
Date::Date(long findate)
{
  year = int(findate/10000.0);
  month = int((findate - year * 10000.0)/100.0);
  day = int(findate - year * 10000.0 - month * 100.0);
}
```

Refer to page 696 for more explanations and examples

Class to Class Conversion

- Class to class conversion same as class to built-in
- Implemented as conversion operator function
  - Uses class name rather than built in data type name
- Requires **forward declaration** of class when class not otherwise already known

Refer to page 699 for more explanations and examples

Class Scope and Duration Categories

- The scope of an identifier defines the portion of a program where the identifier is valid
- There are two categories of scope: local and global
- Each identifier also has a duration: the length of time storage locations are reserved for the variable or function that the identifier names

Class Scope

- Class data members are local to objects created from the class
- An object's data member takes precedence over a global variable of the same name
- Class member functions are global in the file where they're defined but can be called only for objects created from the class

Class Scope (continued)

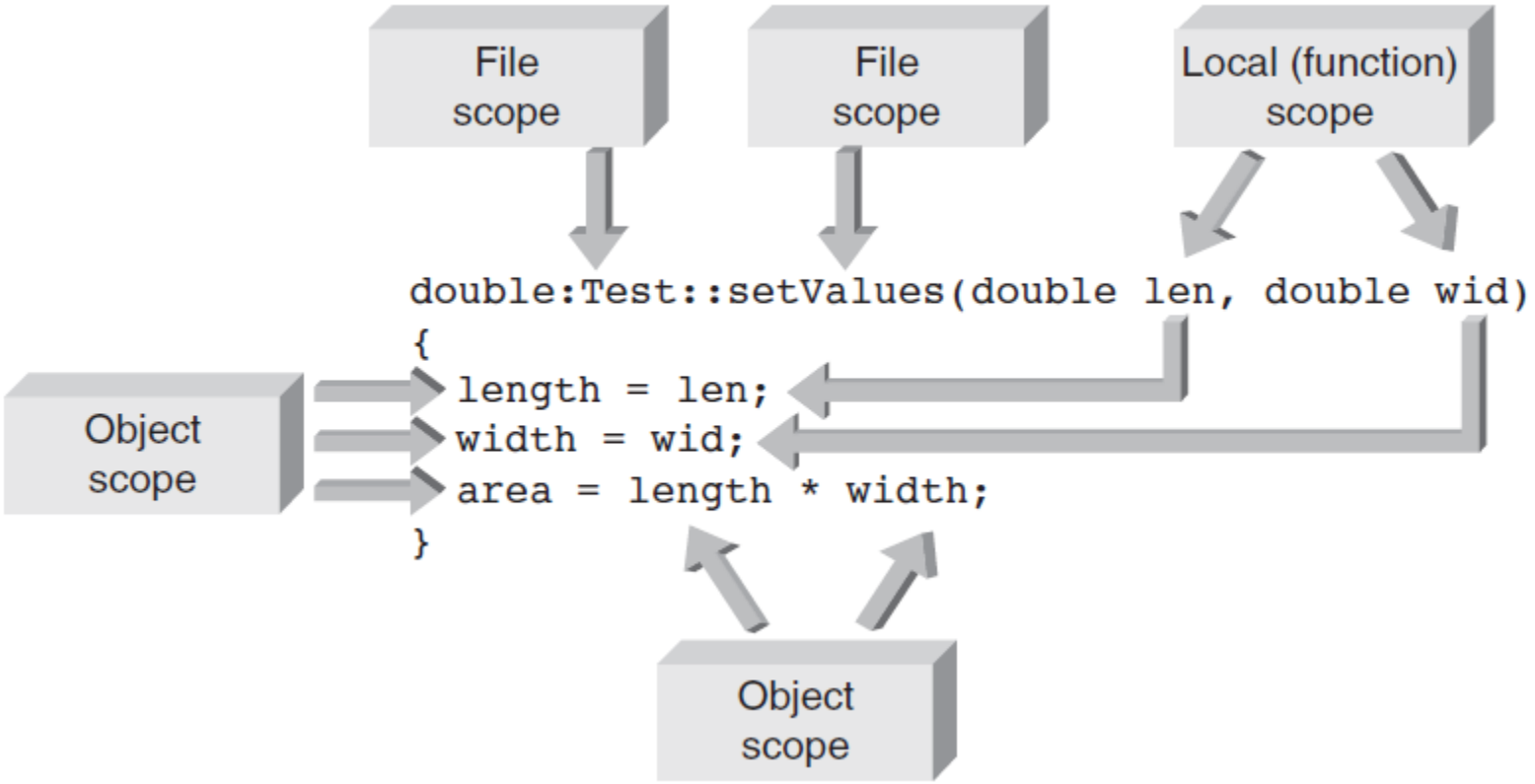


Figure 12.2 Example of scopes

Static Data Members

- As each class object is created, it gets its own block of memory for its data members
- In some cases, it is convenient for every instantiation of a class to share the same memory location for a specific variable

Refer to pages 704,705 for more explanations and examples

Static Class Members (continued)

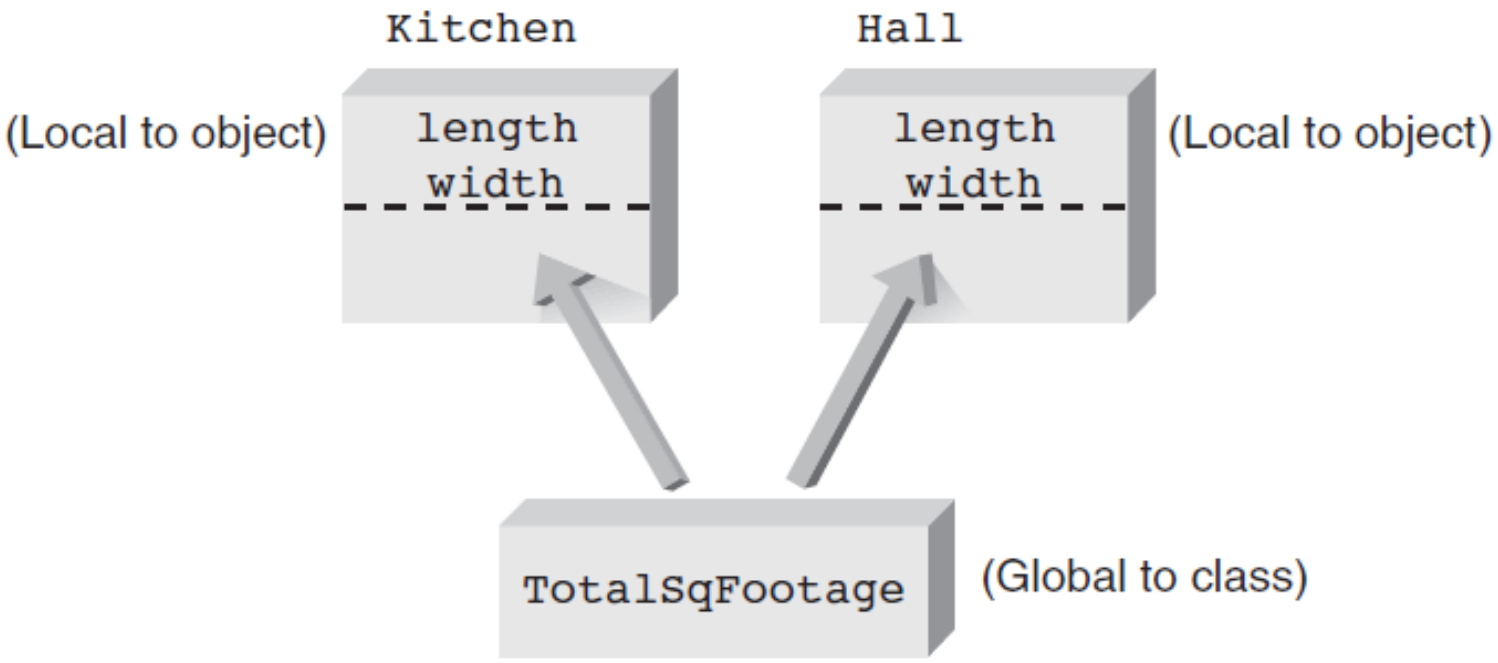


Figure 12.3 Sharing the static data member `TotalSqFootage`

Static Member Functions

- Static member functions can access only static data members and other static member functions
- Their primary purpose is to perform any specialized initialization or operation procedures on static member variables before any object creations

Refer to page 707 for more explanations and examples

Class Inheritance and Polymorphism

- Ability to create new classes from existing ones is the underlying motivation and power behind class- and object-oriented programming techniques
- Inheritance:
  - Deriving one class from another class
- Polymorphism:
  - Redefining how member functions of related classes operate based on the class object being referenced

Class Inheritance and Polymorphism (continued)

- Base class: Initial class used as a basis for a derived class
  - Also called parent or superclass
- Derived class: New class incorporating all the data members and member functions of its base class
  - Also called child class or subclass
  - Can, and usually does, add its own data members and member functions
  - Can override any base class function

Class Inheritance and Polymorphism (continued)

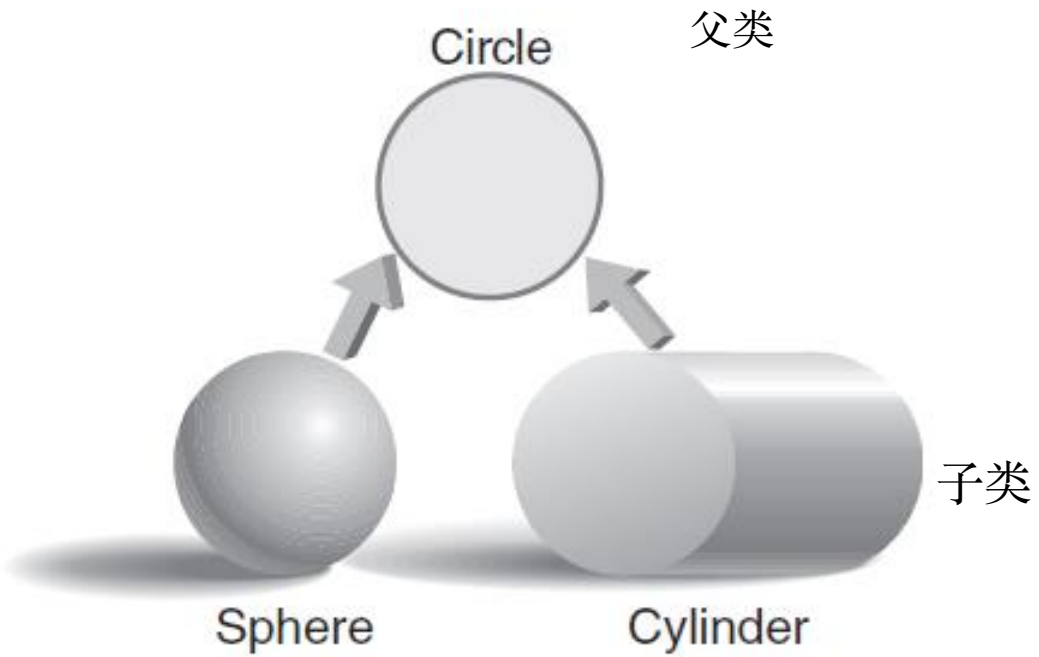


Figure 12.4 Relating object types

Class Inheritance and Polymorphism (continued)

- Simple inheritance: Derived type has only one base type
- Multiple inheritance: Derived type has two or more base types
- Class hierarchies: Illustrate the hierarchy or order in which one class is derived from another
- Derived class has same form as any other class except: Includes access specifier and base class name

子类 父类

```
class derivedClassName : classAccess baseClassName
```

Class Inheritance and Polymorphism (continued)

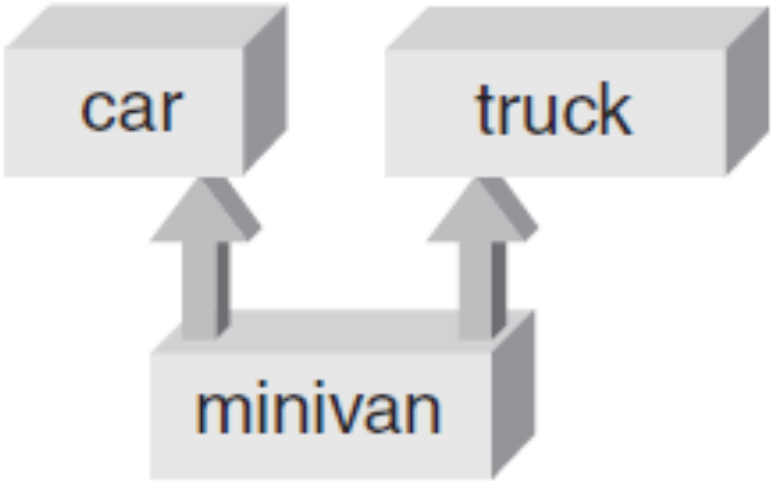


Figure 12.5 An example of multiple inheritance

Class Inheritance and Polymorphism (continued)

- Class derivations are formally called class hierarchies
  - `Circle` is the name of an existing class
  - `Cylinder` can be derived as shown below:

```
class Cylinder : public Circle
{
  // place any additional data members and
  // member functions in here
};  // end of Cylinder class declaration
```

Access Specifications

- Private status ensures that data members can only be accessed by class member functions or friends
  - No access to nonclass functions except friends
- Protected status same as private status except derived classes can access the base class data member

Access Specifications (continued)

Base Class Member	Derived Class Access Specifier	Derived Class Member
private	`: private`	inaccessible
protected	`: private`	private
public	`: private`	private
private	`: public`	inaccessible
protected	`: public`	protected
public	`: public`	public
private	`: protected`	inaccessible
protected	`: protected`	protected
public	`: protected`	protected

Table 12.1 Inherited Access Restrictions

Access Specifications (continued)

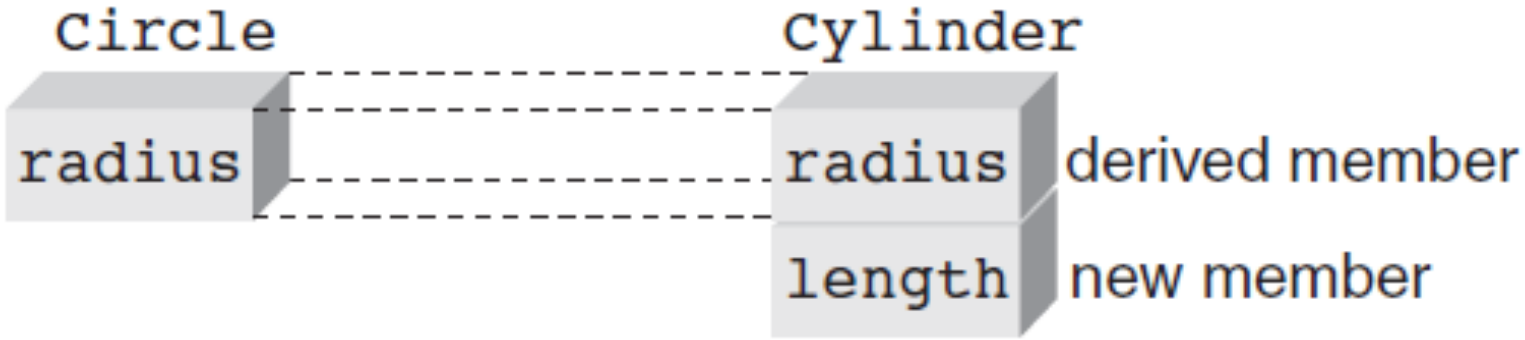


Figure 12.6 Relationship between `Circle` and `Cylinder` data members

Refer to pages 712,713 for more explanations and examples

Access Specifications (continued)

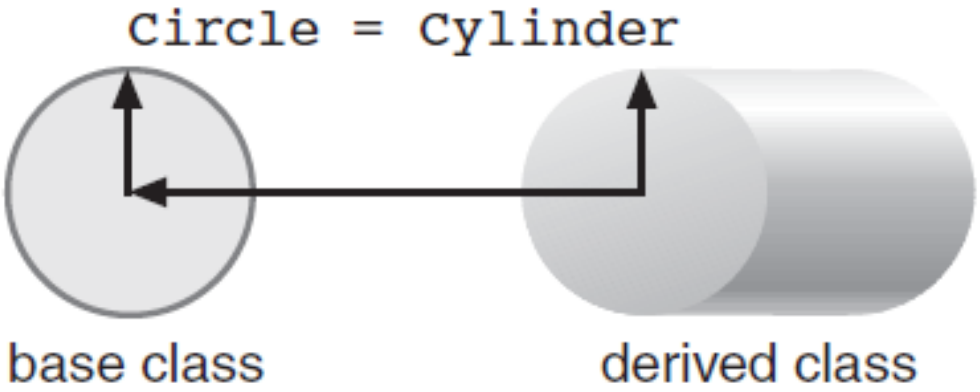


Figure 12.7 An assignment from derived to base class

Refer to pages 714-716 for more explanations and examples

Virtual Functions

- Polymorphism permits using the same function name to invoke:
  - One response in a base class's objects
  - Another response in a derived class's objects

Virtual Functions (continued)

- Two types of function binding
  - Static binding: Determination of function to call is made at compile time
  - Dynamic binding: Determination of function to call is made at runtime based on object type making call
    - Depends on virtual functions
- Virtual function
  - Compiler creates a pointer to a function; assigns value to pointer upon function call

Refer to pages 717-721 for more explanations and examples

Virtual Functions (continued)

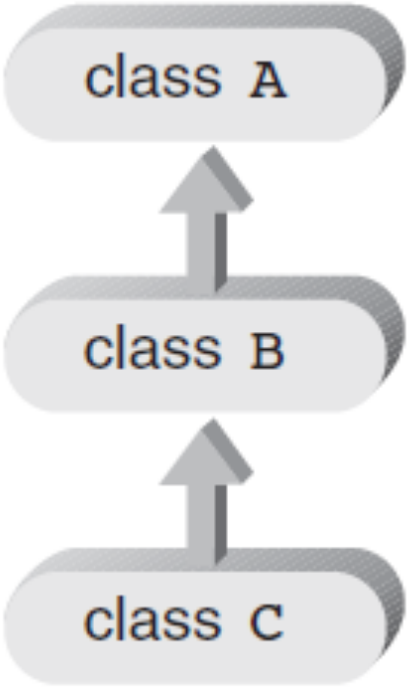


Figure 12.8 An inheritance diagram

Refer to page 721 for more explanations and examples

Common Programming Errors

- Using a `const` reference parameter in both the function prototype and header when overloading the extraction operator, >>
- Using the `static` keyword when defining a `static` data member or member function
- The `static` keyword should be used only when a data member is being declared in the class's declaration section
- Failing to instantiate static data members in a class's implementation section

Common Programming Errors (continued)

- Attempting to make a conversion operator function a friend rather than a member function
- Attempting to specify a return type for a conversion operator function
- Attempting to override a virtual function without using the same type and number of arguments as the original function

Common Programming Errors (continued)

- Using the `virtual` keyword in the class's implementation section
- Functions are declared as virtual only in the class's declaration section

Summary

- The `ostream` class's insertion operator, <<, can be overloaded to display objects.
- The `istream` class's extraction operator, >>, can be overloaded to input values in an object's data members. 重载

Summary (continued)

- Four categories of data type conversions
  - Built-in types to built-in types
  - Class types to built-in types
  - Built-in types to class types
  - Class types to class types
- Type conversion constructor: First argument is not a member of its class; any remaining arguments have default values
- Conversion operator function: Member function having the name of a class
  - No explicit arguments or return type

Summary (continued)

- Data members are local to the objects in which they're created
  - If a global variable name is used in a class, the global variable is hidden by the object's data member of the same name, if one exists
  - In this case, the global variable can be accessed by using the scope resolution operator, `::`
- The scope of all member functions is the file in which they're defined

Summary (continued)

- A static data member is shared by all class objects and provides a means of communication between objects
  - Static data members must be declared in the class's declaration section and are defined outside the declaration section
  - Static member functions can access only static data members and other static member functions
  - They must be declared in the class's declaration section and are defined outside the declaration section

Summary (continued)

- Inheritance: Capability of deriving one class from another class
  - Initial class used as basis for derived class: base, parent, or superclass
  - Derived class: child or subclass
- Base class functions can be overridden by derived class functions with same name
- Polymorphism: Capability of having the same function name invoke different responses based on the object making the call

Summary (continued)

- Override functions and virtual functions can be used to implement polymorphism
- In static binding, the determination of which function is called is made at compile time; in dynamic binding, the determination is made at runtime
- Virtual function: Dynamic binding should take place
  - Specification made in function's prototype by placing keyword `virtual` before the function's return type
  - After a function is declared `virtual` it remains virtual for all derived classes

Homework

- P724 exercises 1, 2