CHAPTER 3

Inheritance and Class Hierarchies

Chapter Objectives

- To understand inheritance and how it facilitates code reuse
- To understand how C++ determines which member function to execute when there are multiple member functions with the same name in a class hierarchy
- To learn how to define and use abstract classes as base classes in a hierarchy
- To understand how to create namespaces and to learn more about visibility
- To learn about multiple inheritance and how to use it effectively
- To become familiar with a class hierarchy for geometric shapes

Introduction to Inheritance and Class Hierarchies

Section 3.1

Introduction to Inheritance and Class Hierarchies

- Object-oriented programming (OOP) enables programmers to reuse previously written code saved as classes
- Code reuse reduces the time required to code new applications
- Previously written code has been tested and debugged which enables new applications to be more reliable
- The C++ Standard Library gives the C++ programmer a head start in developing new applications
- C++ programmers also can build and reuse their own individual libraries of classes

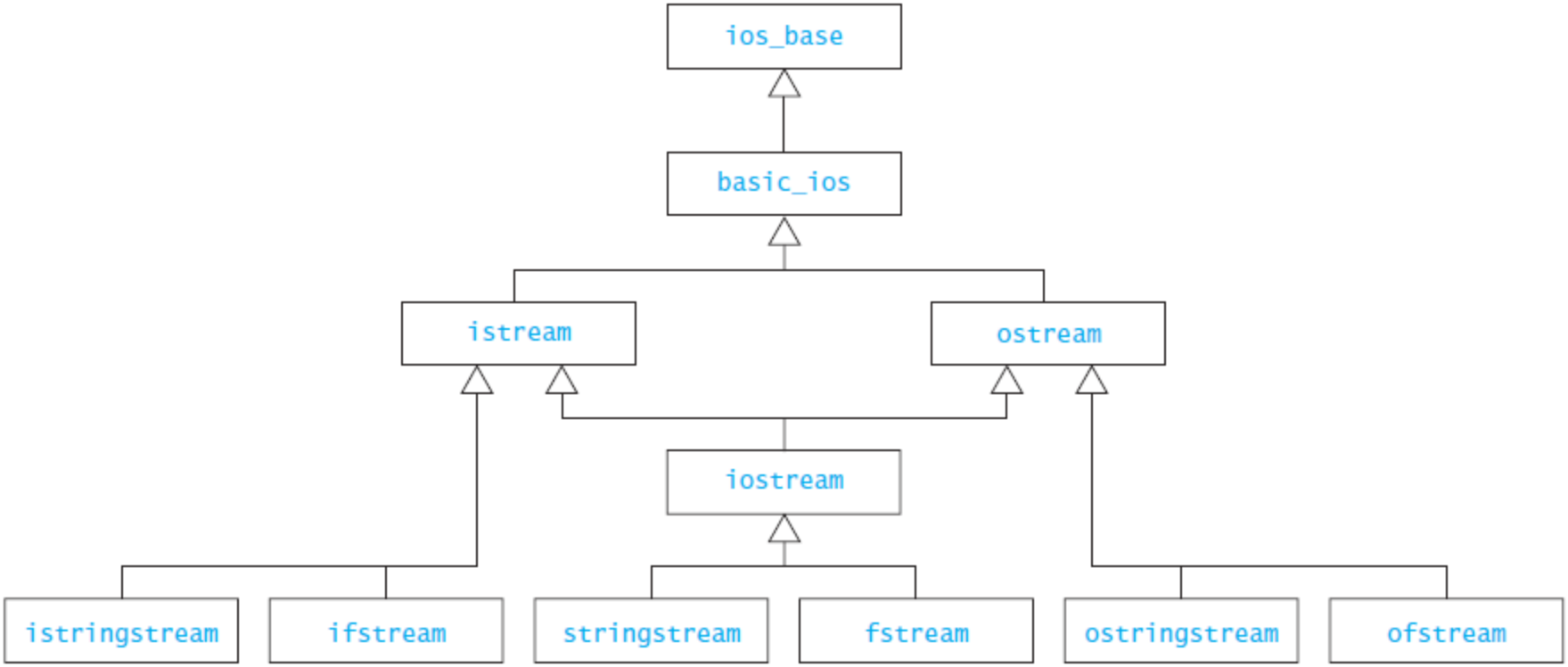
Introduction to Inheritance and Class Hierarchies (cont.)

- In OOP, a programmer can create a similar class by *extending* an existing class, rather than by writing an entirely new class
- The new class (called the *derived class*) can have additional data fields and member functions
- The derived class *inherits* the data fields and member functions of the original class (called the *base class*)
- A *subclass* is another term used for a derived class, and *superclass* is another term used for a base class
- These terms are general, object-oriented terms used in the discussion of other object-oriented programming languages
- C++, however, uses the terms *derived class* and *base class*

Introduction to Inheritance and Class Hierarchies (cont.)

- Classes can be arranged in a hierarchy, with a top-level base class
- Objects farther down the hierarchy are more complex and less general than those farther up
- Inheritance in OOP is analogous to inheritance in humans
  - We all inherit genetic traits from our parents
  - If we are fortunate, we may even have some ancestors who have left us an inheritance of monetary value
  - We benefit from our ancestors' resources, knowledge, and experiences, but our experiences will not affect how our parents or ancestors developed
- C++ classes usually have only one parent, but may have multiple parents

Introduction to Inheritance and Class Hierarchies (cont.)



Is-a Versus *Has-a* Relationships

- The *is-a* relationship between classes means that every instance of one class is also an instance of the other class (but not the other way around)
- A jet airplane is an airplane, but not all airplanes are jet airplanes
- The *is-a* relationship is represented in object oriented programming by extending a class
- To say that a jet plane *is an* airplane means that the jet plane class is a derived class of the airplane class

Is-a Versus *Has-a* Relationships (cont.)

- The *has-a* relationship between classes means that every instance of one class is or may be associated with one or more instances of the other class (but not necessarily the other way around)
  - For example, a jet plane has a jet engine
- The *has-a* relationship is represented by declaring in one class a data field whose type is another class

Is-a Versus *Has-a* Relationships (cont.)

- We can combine *is-a* and *has-a* relationships
  - A jet plane *is* an airplane, and it *has* a jet engine
  - An airplane *has* a tail, so a jet plane does too because it *is* an airplane
- C++ allows you to capture both the inheritance (*is-a*) relationship and the *has-a* relationship:

```
class Jet_Plane : public Airplane {
   private:
      int num_engines;
      Jet_Engine jets[4]; // Jet planes have up to 4 engines.
   // ...
};
```

Is-a Versus *Has-a* Relationships (cont.)

- We can combine *is-a* and *has-a* relationships
  - A jet plane *is* an airplane, and it has a jet
  - An airplane *has* a tail, so a jet plane does airplane
- C++ allows you to capture both the inhe relationship and the *has-a* relationship:

The part of the class heading following the colon specifies that `Jet_Plane` is a derived class of `Airplane`

```
class Jet_Plane : public Airplane {
   private:
      int num_engines;
      Jet_Engine jets[4]; // Jet planes have up to 4 engines.
   // ...
};
```

Is-a Versus *Has-a* Relationships (cont.)

- We can combine *is-a* and *has-a* relationships
  - A jet plane *is* an airplane, and it has a jet engine
  - An airplane *has* a tail, so a jet plane does too because it is an airplane
- C++ allows you to capture both the inhe… relationship and the *has-a* relationship:

```
class Jet_Plane : public Airplane {
   private:
      int num_engines;
      Jet_Engine jets[4]; // Jet planes have up to 4 engines.
   // ...
};
```

The data field `jets` (type `Jet_Engine[4]`) stores information for up to 4 jet engines for a `Jet_Plane` object.

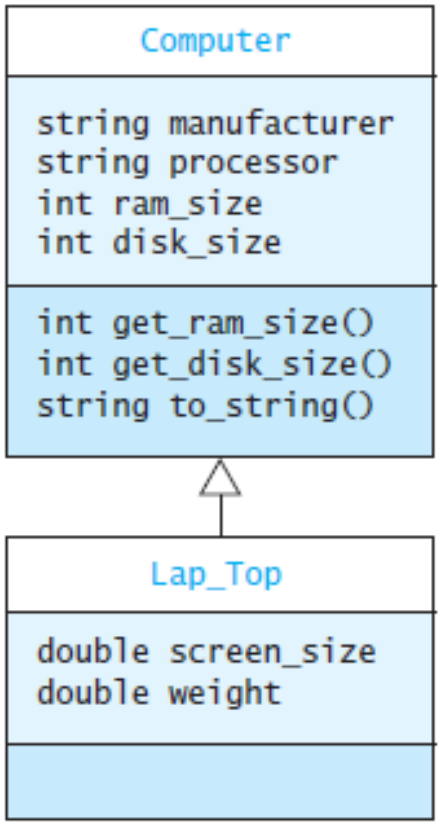
A Base Class and a Derived Class

- A computer has a:
  - manufacturer
  - processor
  - RAM
  - disk

A Base Class and a Derived Class (cont.)

- A laptop computer is a kind of computer, so it has all the properties of a computer plus some additional features:
  - screen size
  - weight
- We can define class `Lap_Top` as a derived class of class `Computer`

A Base Class and a Derived Class (cont.)



Class Computer

Computer.h

```
#ifndef COMPUTER_H_
#define COMPUTER_H_
#include <string>
class Computer {
   private:
      std::string manufacturer;
      std::string processor;
      int ram_size;
      int disk_size;
   public:
      Computer(const std::string& man,
               const std::string proc,
               int ram, int disk) :
         manufacturer(man), processor(proc),
         ram_size(ram), disk_size(disk) {}
      int get_ram_size() const { return ram_size; }
      int get_disk_size() const { return disk_size; }
      // ...
      std::string to_string() const;
};
#endif
```

Computer.cpp

```
#include "Computer.h"
#include <sstream>
using std::ostringstream;
using std::string;
string Computer::to_string() const {
   ostringstream sb;
   sb << "Manufacturer: " << manufacturer
      << "\nCPU: " << processor
      << "\nRAM: " << ram_size << " megabytes"
      << "\nDisk: " << disk_size << " gigabytes";
   return sb.str();
}
```

Class Computer

Computer.h

```
#ifndef COMPUTER_H_
#define COMPUTER_H_
#include <string>
class Computer {
   private:
      std::string manufacturer;
      std::string processor;
      int ram_size;
      int disk_size;
   public:
      Computer(const std::string& man,
               const std::string proc,
               int ram, int disk) :
         manufacturer(man), processor(proc),
         ram_size(ram), disk_size(disk) {}
      int get_ram_size() const { return ram_size; }
      int get_disk_size() const { return disk_size; }
      // ...
      std::string to_string() const;
};
#endif
```

Computer.cpp

```
#include "Computer.h"
                                                st {

                                                facturer

                                                 " megabytes"
      << "\nDisk: " << disk_size << " gigabytes";
   return sb.str();
}
```

We refer to the standard `string` class using the fully qualified name `std::string` in file `Computer.h`

Class Computer

Computer.h

```
#ifndef COMPUTER_H_
#define COMPUTER_H_
#include <string>
class Computer {
   private:
      std::string manufacturer;
      std::string processor;
      int
      int
   public:
      Comp
                                    manufacturer(man), processor(proc),
                                    ram_size(ram), disk_size(disk) {}
      int get_ram_size() const { return ram_size; }
      int get_disk_size() const { return disk_size; }
      // ...
      std::string to_string() const;
};
#endif
```

But we use the abbreviated name `string` in file `Computer.cpp`

Computer.cpp

```
#include "Computer.h"
#include <sstream>
using std::ostringstream;
using std::string;
string Computer::to_string() const {
   ostringstream sb;
   sb << "Manufacturer: " << manufacturer
      << "\nCPU: " << processor
      << "\nRAM: " << ram_size << " megabytes"
      << "\nDisk: " << disk_size << " gigabytes";
   return sb.str();
}
```

Class Computer

Computer.h

```
#ifndef COMPUTER_H_
#define COMPUTER_H_
#include <string>
class Computer {
   private:
      // ...
      std::string to_string() const;
};
#endif
```

We can do this in file `Computer.cpp` because it contains the statement `using string::std;`

The reason we did not put the `using` statement in file `Computer.h` is discussed in Section 3.5.

Computer.cpp

```
#include "Computer.h"
#include <sstream>
using std::ostringstream;
using std::string;
string Computer::to_string() const {
   ostringstream sb;
   sb << "Manufacturer: " << manufacturer
      << "\nCPU: " << processor
      << "\nRAM: " << ram_size << " megabytes"
      << "\nDisk: " << disk_size << " gigabytes";
   return sb.str();
}
```

Class Lap_Top

Lap_Top.h

```
#ifndef LAP_TOP_H_
#define LAP_TOP_H_
#include "Computer.h"
class Lap_Top : public Computer {
  private:
    double screen_size;
    double weight;
  public:
    Lap_Top(const std::string& man, const std::string& proc,
            int ram, int disk, double screen, double wei) :
      Computer(man, proc, ram, disk), screen_size(screen), weight(wei) {}
};
#endif
```

Class Lap_Top

Lap_Top.h

```
#ifndef LAP_TOP_H_
#define LAP_TOP_H_
#include "Computer.h"
class Lap_Top : public Computer {
  private:
    double screen_size;
    double weight;
  public:
    Lap_Top(const std::string& man, const std::string& proc,
            int ram, int disk, double screen, double wei) :
      Computer(man, proc, ram, disk), screen_size(screen), weight(wei) {}
};
#endif
```

This line indicates that class `Lap_Top` is derived from class `Computer` and inherits its member data and member functions

Class Lap_Top

Lap_Top.h

```
#ifndef LAP_TOP_H_
#define LAP_TOP_H_
#include "Computer.h"
class Lap_Top : public Computer {
  private:
    double screen_size;
    double weight;
  public:
    Lap_Top(const std::string& man, cons
            int ram, int disk, double screen, double wei) :
      Computer(man, proc, ram, disk), screen_size(screen), weight(wei) {}
};
#endif
```

If we omit the keyword `public` when we declare a derived class, we will declare a derived class with private inheritance. This means that the derived class will inherit the data fields and functions of the base class, but they will not be visible to client classes.

Initializing Data Fields in a Derived Class

Lap_Top.h

```
#ifndef LAP_TOP_H_
#define LAP_TOP_H_
#include "Computer.h"
class Lap_Top : public Computer {
  private:
    double screen_size;
    double weight;
  public:
    Lap_Top(const std::string& man, const std::string& proc,
            int ram, int disk, double screen, double wei) :
      Computer(man, proc, ram, disk), screen_size(screen), weight(wei) {}
};
#endif
```

The constructor for class `Lap_Top` must begin by initializing the four data fields inherited from class `Computer`. Because those data fields are private to the base class, C++ requires that they be initialized by a base class constructor

Initializing Data Fields in a Derived Class (cont.)

Lap_Top.h

```
#ifndef LAP_TOP_H_
#define LAP_TOP_H_
#include "Computer.h"
class Lap_Top : public Computer {
  private:
    double screen_size;
    double weight;
  public:
    Lap_Top(const std::string& man, const std::string& proc,
            int ram, int disk, double screen, double wei) :
      Computer(man, proc, ram, disk), screen_size(screen), weight(wei) {}
};
#endif
```

We indicate this by inserting a call to the `Computer` constructor as the first initialization expression (following the ":")

Initializing Data Fields in a Derived Class (cont.)

Lap_Top.h

```
#ifndef LAP_TOP_H_
#define LAP_TOP_H_
#include "Computer.h"
class Lap_Top : public Computer {
  private:
    double screen_size;
    double weight;
  public:
    Lap_Top(const std::string& man, const std::string& proc,
            int ram, int disk, double screen, double wei) :
      Computer(man, proc, ram, disk), screen_size(screen), weight(wei) {}
};
#endif
```

The `Computer` constructor (indicated here by its *signature*) is run before the `Lap_Top` constructor

The No-Parameter Constructor

- If the execution of any constructor in a derived class does not invoke a base class constructor, C++ automatically invokes the no-parameter constructor for the base class
- C++ does this to initialize the part of the object inherited from the base class before the derived class starts to initialize its part of the object

The No-Parameter Constructor (cont.)

PITFALL

Not Defining the No-Parameter Constructor

If no constructors are defined for a class, the no-parameter constructor for that class will be provided by default. However, if any constructors are defined, the no-parameter constructor must also be defined explicitly if it needs to be invoked. C++ does not provide it automatically in this case, because it may make no sense to create a new object of that type without providing initial data field values. (It was not defined in class `Lap_Top` or `Computer` because we want the client to specify some information about a `Computer` object when that object is created.) If the no-parameter constructor is defined in a derived class but is not defined in the base class, you will get a syntax error. For example, the g++ compiler displays an error such as

`no matching function call to` *class_name*`::`*class_name*`()`

You can also get this error if a derived-class constructor does not explicitly call a base-class constructor. There will be an implicit call to the no-parameter base-class constructor, so it must be defined.

Protected Visibility for Base-Class Data Fields

- The data fields inherited from class `Computer` have private visibility—they can be accessed only within class `Computer`
- The following assignment statement would not be valid in class `Lap_Top`:

```
manufacturer = man;
```

- C++ provides a less restrictive form of visibility called *protected visibility* to allow a derived class to directly access data fields declared in its base class
- A data field (or member function) with protected visibility can be accessed in the class defining it and in any class derived class from that class
- If class `Computer` had the declaration

```
protected:
string manufacturer;
```

  the assignment statement above would be valid in class `Lap_Top`
- In general, it is better to use private visibility in classes because derived classes may be written by different programmers and it is always good practice to restrict and control access to base-class data fields

Member Function Overriding, Member Function Overloading, and Polymorphism

Section 3.2

Member Function Overriding

```
int main() {
    Computer my_computer("Acme", "Intel P4 2.4", 512, 60);
    Lap_Top your_computer("DellGate", "AMD Athlon 2000", 256, 40, 15.0, 7.5);
    cout << "My computer is :\n" << my_computer.to_string() << endl;
    cout << "\nYour computer is :\n" << your_computer.to_string() << endl;
}
```

Member Function Overriding (cont.)

```
int main() {
    Computer my_computer("Acme", "Intel P4 2.4", 512, 60);
    Lap_Top your_computer("DellGate", "AMD Athlon 2000", 256, 40, 15.0, 7.5);
    cout << "My computer is :\n" << my_computer.to_string() << endl;
    cout << "\nYour computer is :\n" << your_computer.to_string() << endl;
}
```

```
My computer is:
Manufacturer: Acme
CPU: Intel P4 2.4
RAM: 512 megabytes
Disk: 60 gigabytes

Your computer is:
Manufacturer: DellGate
CPU: AMD Athlon 2000
RAM: 256 megabytes
Disk: 40 gigabytes
```

Executing the code above produces the output displayed on the left

Member Function Overriding (cont.)

```
int main() {
    Computer my_computer("Acme", "Intel P4 2.4", 512, 60);
    Lap_Top your_computer("DellGate", "AMD Athlon 2000", 256, 40, 15.0, 7.5);
    cout << "My computer is :\n" << my_computer.to_string() << endl;
    cout << "\nYour computer is :\n" << your_computer.to_string() << endl;
}
```

```
My computer is:
Manufacturer: Acme
CPU: Intel P4 2.4
RAM: 512 megabytes
Disk: 60 gigabytes

Your computer is:
Manufacturer: DellGate
CPU: AMD Athlon 2000
RAM: 256 megabytes
Disk: 40 gigabytes
```

Even though `your_computer` is of type `Lap_Top`, the `Lap_Top` fields are not displayed; the call to `to_string()` calls the `to_string()` method inherited from `Computer`

Member Function Overriding (cont.)

- To show the state of a laptop computer, complete with screen size and weight, we need to define a `to_string` member function for class `Lap_Top`
- If class `Lap_Top` has its own `to_string` member function, it will *override* the inherited member function and will be invoked by the member function call `your_computer.to_string()`

Member Function Overriding (cont.)

```
string Lap_Top::to_string() const {
   ostringstream sb;
   sb << Computer::to_string()
      << "\nScreen Size: " << screen_size
      << "\nWeight: " << weight;
   return sb.str();
}
```

Member Function Overriding (cont.)

```
string Lap_Top::to_string() const {
   ostringstream sb;
   sb << Computer::to_string()
      << "\nScreen Size: " << screen_size
      << "\nWeight: " << weight;
   return sb.str();
}
```

`Lap_Top` cannot access the private data fields of `Computer`. To display them, we call `Computer`'s `to_string` function

Member Function Overriding (cont.)

```
string Lap_Top::to_string() const {
   ostringstream sb;
   sb << Computer::to_string()
      << "\nScreen Size: " << screen_size
      << "\nWeight: " << weight;
   return sb.str();
}
```

and then append `Lap_Top`'s data fields

Member Function Overloading

- If we decide to standardize and purchase our laptop computers from only one manufacturer we could introduce a new constructor with one less parameter:

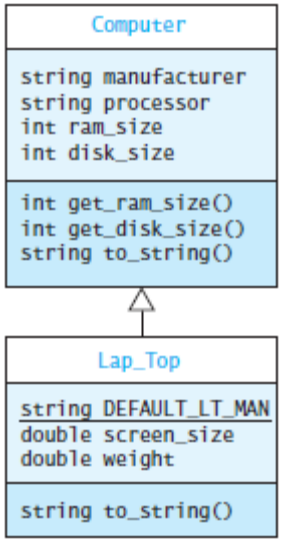
```
Lap_Top(const std::string& proc, int ram, int disk,
        double screen, double wei) :
    Computer(DEFAULT_LT_MAN, proc, ram, disk),
            screen_size(screen), weight(wei) {}
```

- Having multiple member functions with the same name but different signatures in a class is called member function *overloading*

Member Function Overloading (cont.)

- Both of the following statements are now valid:

```
Lap_Top ltp1("Intel P4 2.8", 256,
             40, 14, 6.5);
Lap_Top ltp2("MicroSys",
             "AMD Athlon 2000",
             256, 40, 15, 7.5);
```



- Since it is not specified, the manufacturer string for `ltp1` will be assigned as `DEFAULT_LT_MAN`

Member Function Overloading (cont.)

```
#ifndef LAP_TOP_H_
#define LAP_TOP_H_

#include "Computer.h"

/** Class to represent a laptop computer. */

class Lap_Top : public Computer {
  public:
    // Functions
    /** Construct a Lap_Top object.
        @param man The manufacturer
        @param proc The processor
        @param ram The RAM size
        @param disk The disk size
        @param screen The screen size
        @param wei The weight
    */
    Lap_Top(const std::string& man, const std::string& proc,
            int ram, int disk, double screen, double wei) :
      Computer(man, proc, ram, disk), screen_size(screen), weight(wei) {}

    /** Construct a Lap_Top object with a default manufacturer. */
    Lap_Top(const std::string& proc, int ram, int disk,
            double screen, double wei) :
      Computer(DEFAULT_LT_MAN, proc, ram, disk), screen_size(screen),
               weight(wei) {}

    /** Generate a string representation of a Lap_Top object. */
    std::string to_string() const;

  private:
    // Data Fields
    static const char* DEFAULT_LT_MAN;
    double screen_size;
    double weight;
};

#endif
```

Member Function Overloading (cont.)

```
/** Implementation of the Lap_Top class. */
#include "Lap_Top.h"
#include <sstream>

using std::string;
using std::ostringstream;

string Lap_Top::to_string() const {
  ostringstream sb;
  sb << Computer::to_string()
     << "\nScreen Size: " << screen_size
     << "\nWeight: " << weight;
  return sb.str();
}

const char* Lap_Top::DEFAULT_LT_MAN = "MyBrand";
```

Virtual Functions and Polymorphism

- The pointer variable

  `Computer* the_computer;`

  can be used to reference an object of either type `Computer` or type `Lap_Top`
- In C++, a pointer variable of a baseclass type (general) can point to an object of a derived-class type (specific)
  - `Lap_Top` objects are `Computer` objects with more features
- Suppose you have purchased a laptop computer. What happens when the following statements are executed?

```
the_computer = new Lap_Top("Bravo", "Intel P4 2.4", 256, 40, 15.0, 7.5);
cout << the_computer->to_string() << endl;
```

- What will this function call return:

  `the_computer->to_string()`

  a string with all six data fields for a `Lap_Top` object (using `Lap_Top::to_string`) or just the four data fields defined for a `Computer` object (using `Computer::to_string`)?

Virtual Functions and Polymorphism (cont.)

- The answer is: a string with just the four data fields because `the_computer` is type pointer-to-`Computer`

```
Manufacturer: Bravo
CPU: Intel P4 2.4
RAM: 256 megabytes
Disk: 40 gigabytes
```

- In C++ it is possible for the type of the object receiving the `to_string` message to determine which `to_string` member function is called
- We can do so by changing the declaration of the function `to_string` in the class `Computer` (in `Computer.h`) to a *virtual function*:

```
virtual std::string to_string() const;
```

- If a member function is declared **virtual** when it is called through a pointer (or reference) variable the actual member function will be determined at run time and based on the type of the object pointed to (or referenced)

Virtual Functions and Polymorphism (cont.)

- If the variable `the_computer` points to a `Lap_Top` object and the `to_string` function is declared as virtual, then the expression

  `the_computer->to_string()`

  will call the member function `to_string` defined by the `Lap_Top` class

Virtual Functions and Polymorphism (cont.)

- The feature we just illustrated is an important concept in OOP called *polymorphism*
- Polymorphism is *the quality of having many forms or many shapes*
- Polymorphism enables the program to determine which member function to invoke at run time
- In our example, at compile time, the C++ compiler cannot determine what type of object `the_computer` will point to (type `Computer` or its derived type `Lap_Top`)
- At run time the program knows the type of the object that receives the `to_string` message and calls the appropriate `to_string` function

Virtual Functions and Polymorphism (cont.)

SYNTAX **Declaring a Pointer Type**

FORM:

type-name variable-name*;

EXAMPLE:

```
Computer* the_computer;
```

MEANING:

The variable *variable-name* is declared to be of type pointer-to-*type-name*. As discussed in the *C++ Primer*, a pointer is an object that contains the address of another object. To reference the object pointed to by a pointer variable, use either the dereferencing operator * (for example, `*the_computer`) or the pointer class member access operator -> (for example, `the_computer->`). Pointers are initialized using either the address-of operator (&) or the `new` operator.

Virtual Functions and Polymorphism (cont.)

SYNTAX **Creating New Objects Using the new Operator**

FORM:

```
new type-name;
new type-name(argument-list);
```

EXAMPLE:

```
new Computer("Acme", "Intel P4 2.4", 512, 60);
```

MEANING:

Memory is dynamically allocated for a new object of type *type-name* and the constructor is then called to initialize the object. A pointer to the newly created object is the value of this expression.

Virtual Functions and Polymorphism (cont.)

SYNTAX **Accessing an Object Through a Pointer (the Dereferencing Operator *)**

FORM:

**pointer-variable*

EXAMPLE:

```
*the_computer
```

MEANING:

The expression **pointer-variable* is a reference to the object pointed to by *pointer-variable*. When used on the right-hand side of an assignment, it is the value of the object:

```
Computer my_computer = *the_computer;
```

Now, `my_computer` contains a copy of the `Computer` object referenced by the pointer variable `the_computer`.

When used on the left-hand side of an assignment

```
*the_computer = your_computer;
```

then the referenced object is assigned the value of the right-hand side.

Virtual Functions and Polymorphism (cont.)

SYNTAX **Accessing a Class Member through the Class Member Access Operator**

FORM:

pointer-variable->member-name

EXAMPLE:

```
the_computer->to_string()
my_ram = the_computer->ram_size;
the_computer->ram_size = 1024;
```

MEANING:

The expression *pointer-variable->member-name* is a reference to the member of the object pointed to by *pointer-variable*. If *member-name* is a function, then that function is called. If *member-name* is a data field, then when the expression is used on the right-hand side of an assignment, it is the value of the data field. When the expression is used on the left-hand side of an assignment, then the referenced data field is assigned the value of the right-hand side.

The expression *pointer-variable->member-name* is equivalent to

(**pointer-variable*) .*member-name*

but the -> operator is considered easier to read. Also, the -> operator can be overloaded, but the dot operator cannot.

Virtual Functions and Polymorphism (cont.)

SYNTAX **Virtual Function Definition**

FORM:

`virtual` *function-declaration*`;`
`virtual` *function-declaration* `{` *function-body* `}`

EXAMPLE:

```
virtual std::string to_string() const;
```

MEANING:

If the function declared by *function-declaration* is overridden in a derived class, and if this function is called through a pointer to (or reference to) the base class, then the function body associated with the actual object pointed to (or referenced) will be executed. If a function is declared virtual in a base class, it continues to be virtual in all derived classes. It is, however, a good practice to include the `virtual` declaration in the derived classes too, for documentation purposes.

Virtual Functions and Polymorphism (cont.)

- If we declare the array `lab_computers` as follows:

  ```
  Computer* lab_computers[10];
  ```

  we can store pointers to 10 `Computer` or `Lap_Top` objects in array `lab_computers`
- We can use the following loop to display the information about each object in the array:

  ```
  for (int i = 0; i < 10; i++)
  cout << lab_computers[i]->to_string() << endl;
  ```
- Because of polymorphism, the actual type of the object pointed to by `lab_computers[i]` determines which `to_string` member function will execute (`Computer::to_string` or `Lap_Top::to_string`) for each value of subscript `i`

Virtual Functions and Polymorphism (cont.)

- We can define the `ostream` extraction operator for the Computer class as follows:

```
ostream& operator<<(ostream& os, const Computer& comp) {
   return os << comp.to_string;
}
```

- Because `comp` is a reference to `Computer`, when this function is called with a `Lap_Top` object, the `Lap_Top::to_string` member function will be called
- When it is called with a `Computer` object, the `Computer::to_string` member function will be called

Abstract Classes, Assignment, and Casting in a Hierarchy

Section 3.3

Abstract Classes, Assignment, and Casting in a Hierarchy

- An *abstract class* differs from an actual class (sometimes called a concrete class) in two respects:
  - An abstract class cannot be instantiated
  - An abstract class declares at least one abstract member function, which must be defined in its derived classes
- An *abstract function* is a virtual function that is declared but for which no body (definition) is provided

Abstract Classes, Assignment, and Casting in a Hierarchy (cont.)

- We use an abstract class in a class hierarchy when we need a base class for two or more actual classes that share some attributes
- We may want to declare some of the attributes and define some of the functions that are common to these base classes
- We may also want to require that the actual derived classes implement certain functions
- We can accomplish this by declaring these functions abstract, which makes the base class abstract as well

Abstract Classes, Assignment, and Casting in a Hierarchy (cont.)

- As an example, the Food Guide Pyramid provides a recommendation of what to eat each day based on established dietary guidelines
- There are six categories of foods in the pyramid:
  - fats, oils, and sweets
  - meats, poultry, fish, and nuts
  - milk, yogurt, and cheese
  - vegetables
  - fruits
  - bread, cereal, and pasta
- If we want to model the Food Guide Pyramid, we might make each of these an actual derived classes of an abstract class called `Food` as shown in the following listing:

Abstract Classes, Assignment, and Casting in a Hierarchy (cont.)

Food.h

```
#ifndef FOOD_H_
#define FOOD_H_
class Food {
   private:
      double calories;
   public:
      virtual double percent_protein() const = 0;
      virtual double percent_fat() const = 0;
      virtual double percent_carbohydrates() const = 0;

      double get_calories() const { return calories; }
      void set_calories(double cal) {
         calories = cal;
      }
};
#endif
```

Abstract Classes, Assignment, and Casting in a Hierarchy (cont.)

```
Food.h
#ifndef FOOD_H_
#define FOOD_H_
class Food {
   private:
      double calories;
   public:
      virtual double percent_protein() const = 0;
      virtual double percent_fat() const = 0;
      virtual double percent_carbohydrates() const = 0;

      double get_calories() const { return calories; }
      void set_calories(double cal) {
         calories = cal;
      }
};
#endif
```

These three virtual functions impose the requirement that all derived classes implement these three functions. We would expect a different function definition for each kind of food.

Abstract Classes, Assignment, and Casting in a Hierarchy (cont.)

SYNTAX **Abstract Function Definition**

FORM:

```
virtual function-declaration = 0;
```

EXAMPLE:

```
virtual double percent_protein() const = 0;
```

INTERPRETATION:

The function declared by *function-declaration* is declared to be an abstract function. The expression `= 0;` is given in place of the function body and must be specified in the class declaration. You may also see the term *pure virtual function* to refer to an abstract function.

Referencing Actual Objects

- Because class `Food` is abstract, we can't create type `Food` objects
- The statement

  ```
  Food my_snack();
  ```

  is not valid
- We can use a type `Food` pointer variable to point to an actual object that belongs to a class derived from `Food`
- The following statement creates a `Vegetable` object (derived from `Food`) that is referenced by variable `my_snack` (type pointer-to-`Food`):

  ```
  Food* my_snack = new Vegetable("carrot sticks");
  ```

Summary of Features of Actual Classes and Abstract Classes

Property	Actual Class	Abstract Class
Instances (objects) of this type can be created	Yes	No
This can define instance variables and functions	Yes	Yes
This can define constants	Yes	Yes
The number of these a class can extend	Any number	Any number
This can extend another class	Yes	Yes
This can declare abstract member functions	No	Yes
Pointers to this type can be declared	Yes	Yes
References of this type can be declared	Yes	Yes

Assignments in a Class Hierarchy

- C++ is what is known as a *strongly typed language*
- Each operand has a type and operations can be performed only on operands of the same or compatible types
- This includes the assignment operation:

  l-value = *r-value*

- For the built-in types the r-value (right value) must be of the same type as the l-value (left value) or there must be a conversion defined to convert the r-value into a value that is the same type as the l-value

Assignments in a Class Hierarchy (cont.)

- For class types, the assignment operator may be overridden and overloaded (this is discussed in the following section)
- There is an exception to this rule for pointer types:
  - A pointer variable (l-value) that is of type pointer-to-*base class* may point to a derived object (r-value)
- However, the opposite is not legal

```
Computer* a_computer = new Lap_Top( ... ); // Legal
Lap_Top* a_laptop = new Computer( ... ); // Incompatible
                                         types
```

Casting in a Class Hierarchy

- C++ provides a mechanism, *dynamic casting*, that enables us to process the object referenced by `a_computer` through a pointer variable of its actual type, instead of through a type pointer-to-`Computer`
- The expression

  `dynamic_cast<Lap_Top*>(a_computer)`

  casts the type of the object pointed to by `a_computer` (type `Computer`) to type `Lap_Top`
- The casting operation succeeds only if the object referenced by `a_computer` is, in fact, type `Lap_Top`; if not, the result is the null pointer

Casting in a Class Hierarchy (cont.)

- Casting gives us a type `Lap_Top` pointer to the object that can be processed just like any other type `Lap_Top` pointer
- The expression

  ```
  dynamic_cast<Lap_Top*>(a_computer)->get_weight()
  ```

  will compile because now `get_weight` is applied to a type `Lap_Top` reference
- Similarly, the assignment statement

  ```
  Lap_Top* a_laptop = dynamic_cast<Lap_Top*>(a_computer);
  ```

  is valid because a type `Lap_Top` pointer is being assigned to `a_laptop` (type `Lap_Top*`)
- Keep in mind that the casting operation does not change the object pointed to by `a_computer`; instead, it creates a type `Lap_Top` pointer to it (This is called an *anonymous* or *unnamed pointer*)

Casting in a Class Hierarchy (cont.)

- Using the type `Lap_Top` pointer, we can invoke any member function of class `Lap_Top` and process the object just like any other type `Lap_Top` object
- The cast

```
dynamic_cast<Lap_Top*>(a_computer)
```

  is called a *downcast* because we are casting from a higher type in the inheritance hierarchy (`Computer`) to a lower type (`Lap_Top`)
- Microsoft compilers require you to set a compiler switch to use `dynamic_cast`

Casting in a Class Hierarchy (cont.)

SYNTAX **Dynamic Cast**

FORM:

`dynamic_cast<`*class-name*`*>(`*object-name*`)`

EXAMPLE:

```
dynamic_cast<Lap_Top*>(a_computer)
```

INTERPRETATION:

A pointer of type *class-name* is created to object *object-name*, provided that *object-name* is an actual object of type *class-name* (or a derived class of *class-name*). If *object-name* is not type *class-name*, the result is a null pointer.

CASE STUDY: Displaying Addresses for Different Countries

- **PROBLEM**:
  - Different countries use different conventions when addressing mail
  - We want to write a program that will display the address in the correct format depending on the destination
  - Specifically, we want to distinguish between addresses in Germany and in the United States and Canada
  - In Germany the building number follows the street name (line 1 below), and the postal code (equivalent to the U.S. ZIP code) is placed before the municipality name (line 2 below)
    - This is the opposite of the convention used in the United States and Canada. For example, a typical German address would read

      ```
      Bahnhofstr 345
      D-99999 Jedenstadt
      ```
  - while a typical U.S. and Canadian address would read

    ```
    123 Main St
    Somecity ZZ 99999-9999
    ```
  - However, the Canadian postal code is in a different format

CASE STUDY: Displaying Addresses for Different Countries (cont.)

- **ANALYSIS:**
  - Several components make up an address, and there are numerous variations between countries as to how to arrange them
  - One simple approach would be to ignore these variations and merely record complete address lines as strings
  - However, this approach does not allow us to analyze or validate addresses easily
  - By parsing the addresses into their components, we can validate addresses against databases provided by the national postal authorities and generate address labels that conform to national postal authority standards
  - There are several commercial software packages and databases that do this

CASE STUDY: Displaying Addresses for Different Countries (cont.)

- **ANALYSIS (cont.):**
  - The number of components in an address and their names vary among countries
  - We simplify this situation for the purposes of this case study
  - An address will consist of the following data elements:
    - the house number
    - the street name
    - the municipality
    - the state or province
    - the postal code

CASE STUDY: Displaying Addresses for Different Countries (cont.)

- **ANALYSIS (cont.)**:

  A German address is organized as follows:

  ```
  <street name><house number>
  <postal code><municipality>
  ```

  A U.S./Canadian address, on the other hand, is organized this way:

  ```
  <house number><street name>
  <municipality><state or province><postal code>
  ```

  Because we are originating the mail from the United States, the German address will have the additional requirement that the name of the destination country (Germany) be placed on a third line below the line containing the postal code and municipality

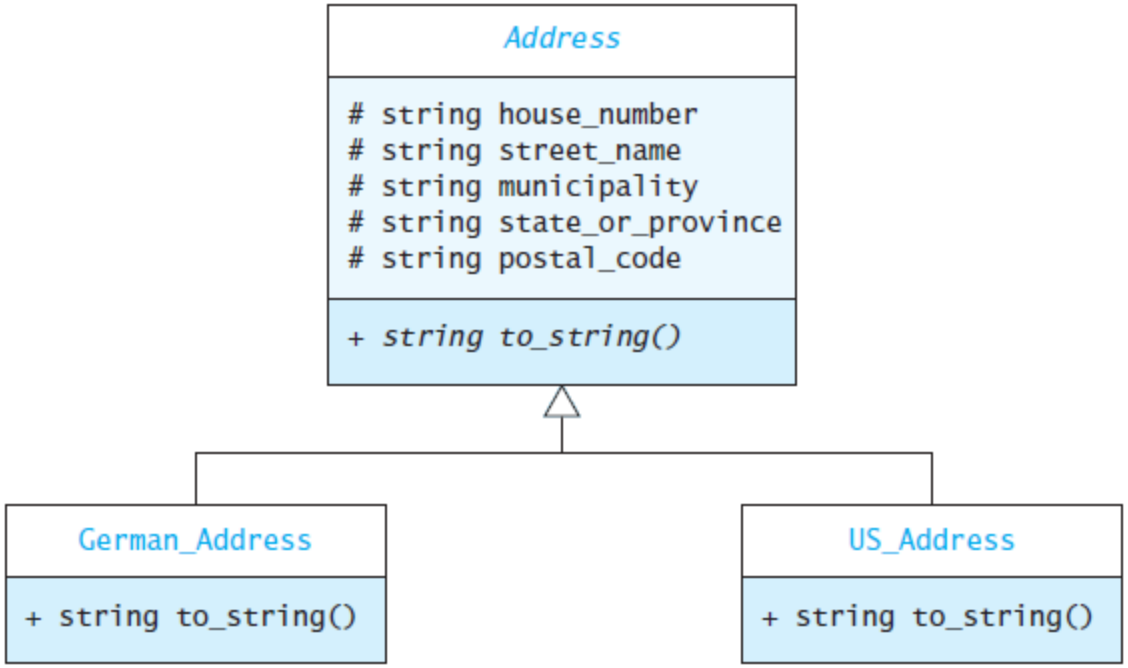
CASE STUDY: Displaying Addresses for Different Countries (cont.)

- **DESIGN**:

  We will create an abstract class, `Address`, that will contain the data elements
- This class will also have the abstract function `to_string`, which will require the derived classes to organize the elements into a format required for the destination country
- The derived classes `German_Address` and `US_Address` will then implement the `to_string` function; the next slide shows the UML diagram
  - The # next to an attribute indicates that it has `protected` visibility

CASE STUDY: Displaying Addresses for Different Countries (cont.)

- **DESIGN (cont.):**



CASE STUDY: Displaying Addresses for Different Countries (cont.)

- **IMPLEMENTATION:**

```
#ifndef ADDRESS_H_
#define ADDRESS_H_

#include <string>

/** Declaration of the abstract class Address. */
class Address {
  public:
    // Functions
    Address(const std::string& hn, const std::string& str,
            const std::string muni, const std::string& stpro,
            const std::string& pc) :
      house_number(hn), street_name(str), municipality(muni),
          state_or_province(stpro), postal_code(pc) {}

    virtual std::string to_string() const = 0;

    virtual ~Address() {}

    void set_house_number(std::string new_house_number) {
      house_number = new_house_number;
    }

  protected:
    // Data Fields
    std::string house_number;
    std::string street_name;
    std::string municipality;
    std::string state_or_province;
    std::string postal_code;
};

/** Declaration of the class German_Address. */
class German_Address : public Address {
  public:
    German_Address(const std::string& hn, const std::string& str,
                   const std::string& muni, const std::string& pc) :
      Address(hn, str, muni, "", pc) {}

    std::string to_string() const;
};

/** Declaration of the class US_Address. */
// Exercise

#endif
```

CASE STUDY: Displaying Addresses for Different Countries (cont.)

- **IMPLEMENTATION:**

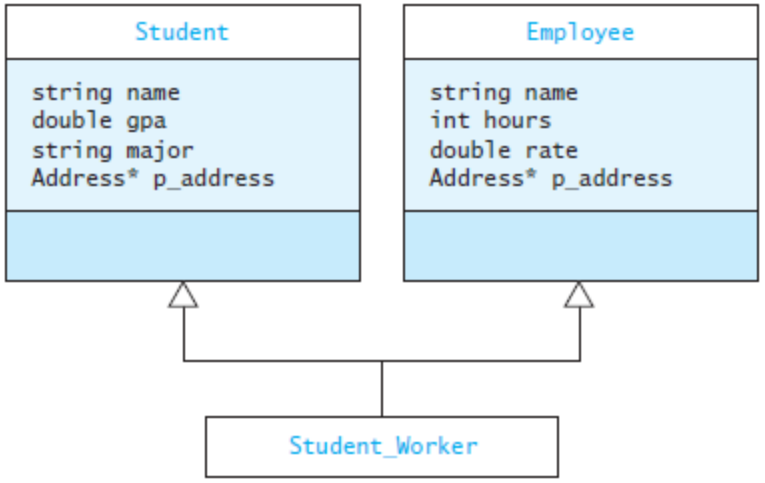
```
/** Implementation file for the class German_Address */
#include "Address.h"
#include <sstream>
using std::string;
using std::ostringstream;

string German_Address::to_string() const {
  ostringstream result;
  result << street_name << " " << house_number << "\n"
         << postal_code << " " << municipality << "\n"
         << "Germany\n";
  return result.str();
}
```

Multiple Inheritance

Section 3.4

Multiple Inheritance

- *Multiple inheritance* refers the ability of a class to extend more than one class
- For example, a university has many students who are full-time students and many employees who are full-time employees, but also some student employees (who are both students and employees)



Multiple Inheritance (cont.)

- In multiple inheritance, all the data fields for the derived class are inherited from its base classes
- Multiple inheritance is a language feature that can lead to ambiguity; if a class extends two classes, and each declares the same data field (for example, `name` in `Student` and `Employee`), which one does the derived class inherit?
- In C++, the answer is *both*
- We will show how C++ handles this
- (Because of the ambiguity problem, the Java programming language does not support multiple inheritance)

Definition of Student_Worker

Student_Worker.h

```
#ifndef STUDENT_WORKER_H_
#define STUDENT_WORKER_H_
#include "Employee.h"
#include "Student.h"
class Student_Worker : public Employee, public Student {
   public:
      Student_Worker(const std::string& the_name,
                     Address* the_address,
                     double the_rate,
                     const std::string& the_major) :
         Employee(the_name, the_address, the_rate),
         Student(the_name, the_address, the_major) {}
      std::string to_string() const;
};
#endif
```

Definition of `Student_Worker` (cont.)

Student_Worker.h

```
#ifndef STUDENT_WORKER_H_
#define STUDENT_WORKER_H_
#include "Employee.h"
#include "Student.h"
class Student_Worker : public Employee, public Student {
   public:
      Student_Worker(const std::string& the_name,
                     Address* the_address,
                     double the_rate,
                     const std::string& the_major) :
         Employee(the_name, the_address, the_rate),
         Student(the_name, the_address, the_major) {}
      std::string to_string() const;
};
#endif
```

The heading shows that class `Student_Worker` extends class `Employee` and class `Student`

Definition of `Student_Worker` (cont.)

```
Student_Worker.h

#ifndef STUDENT_WORKER_H_
#define STUDENT_WORKER_H_
#include "Employee.h"
#include "Student.h"
class Student_Worker : public Employee, public Student
   public:
      Student_Worker(const std::string& the_name,
                     Address* the_address,
                     double the_rate,
                     const std::string& the_major) :
         Employee(the_name, the_address, the_rate),
         Student(the_name, the_address, the_major) {}
      std::string to_string() const;
};
#endif
```

The constructor contains two initialization expressions separated by a comma

Definition of `Student_Worker` (cont.)

Student_Worker.h

```
#ifndef STUDENT_WORKER_H_
#define STUDENT_WORKER_H_
#include "Employee.h"
#include "Student.h"
class Student_Worker : public Employee, public Student {
   public:
      Student_Worker(const std::string& the_name,
                     Address* the_address,
                     double the_rate,
                     const std::string& the_major) :
         Employee(the_name, the_address, the_rate),
         Student(the_name, the_address, the_major) {}
      std::string to_string() const;
};
#endif
```

The first initializes the `Employee` part

Definition of `Student_Worker` (cont.)

```
Student_Worker.h

#ifndef STUDENT_WORKER_H_
#define STUDENT_WORKER_H_
#include "Employee.h"
#include "Student.h"
class Student_Worker : public Employee, public Student {
   public:
      Student_Worker(const std::string& the_name,
                     Address* the_address,
                     double the_rate,
                     const std::string& the_major) :
          Employee(the_name, the_address, the_rate),
          Student(the_name, the_address, the_major) {}
      std::string to_string() const;
};
#endif
```

The second initializes the `Student` part

Definition of `Student_Worker` (cont.)

Student_Worker.h

```
#ifndef STUDENT_WORKER_H_
#define STUDENT_WORKER_H_
#include "Employee.h"
#include "Student.h"
class Student_Worker : public Employee, public Student {
   public:
      Student_Worker(const std::string& the_name,
                     Address* the_address,
                     double the_rate,
                     const std::string& the_major) :
         Employee(the_name, the_address, the_rate),
         Student(the_name, the_address, the_major) {}
      std::string to_string() const;
};
#endif
```

There is no argument given for data fields `hours` or `gpa`, so they are initialized to default values

Definition of `Student_Worker` (cont.)

Student_Worker.cpp

```
/** Return string representation of Student_Worker. */
std::string Student_Worker::to_string() const {
   std::ostringstream result;
   result << name << '\n'
          << p_address->to_string()
          << "\nMajor: " << major
          << " GPA: " << gpa
          << " rate: " << rate
          << " hours: " << hours;
   return result.str();
}
```

Definition of `Student_Worker` (cont.)

Student_Worker.cpp

```
/** Return string representation of Student_Worker. */
std::string Student_Worker::to_string() const {
   std::ostringstream result;
   result << name << '\n'
          << p_address->to_string()
          << "\nMajor: " << major
          << " GPA: " << gpa
          << " rate: " << rate
          << " hours: " << hours;
   return result.str();
}
```

When we attempt to compile `Student_Worker.cpp`, we receive an error message that says that the references to `name` and `p_address` are ambiguous

Definition of `Student_Worker` (cont.)

Student_Worker.cpp

```
/** Return string representation of Student_Worker. */
std::string Student_Worker::to_string() const {
   std::ostringstream result;
   result << Student::name << '\n'
          << Student::p_address->to_string()
          << "\nMajor: " << major
          << " GPA: " << gpa
          << " rate: " << rate
          << " hours: " << hours;
   return result.str();
}
```

We can fix this by modifying the code as shown

Definition of `Student_Worker` (cont.)

Student_Worker.cpp

```
/** Return string representation of Student_Worker. */
std::string Student_Worker::to_string() const {
   std::ostringstream result;
   result << Student::name << '\n'
          << Student::p_address->to_string()
          << "\nMajor: " << major
          << " GPA: " << gpa
          << " rate: " << rate
          << " hours: " << hours;
   return result.str();
}
```

While this resolved the error, it is not a very good solution. We still effectively have two `name` data fields and two `p_address` data fields, even though the constructor sets them both to the same value. Because the two `p_address` fields are the same, we may get a run-time error when the destructor for `Student_Worker` is called

Refactoring the Employee and Student Classes

- A better solution is to recognize that both `Employees` and `Students` have common data fields
- These common data fields and their associated member functions could be collected into a separate class, which would become a common base class `Person`
- This process is known as *refactoring* and is often used in object-oriented design
- However, it leads to similar problems, because there are two `Person` components in the `Student_Worker` class: one inherited from `Employee` and the other inherited from `Student`
- A workable solution to this problem is beyond the scope of the chapter
  - you can find this solution in Appendix A.4

Namespaces and Visibility

Section 3.5

Namespaces

- The entire C++ standard library is contained in the namespace `std`
- Namespaces are used to group collections of declarations into a functional unit
- By using namespaces, clashes between duplicate names are avoided
  - An adventure game program may define the class `map` to represent the area where the action takes place
  - The standard library also defines a class called `map`, which associates values with keys (discussed in Chapter 9)
  - To avoid clashes between the two `map` classes, we prefix the namespace name before the class name

Namespaces (cont.)

- This declaration of a class map uses the standard map as a component:

```
#include <map> // Get defn of standard map
class map {
   private:
      std::map the_map; // Declare a component.
   ...
};
```

Declaring a Namespace

- We could make the distinction even clearer by defining our own namespace `game` and placing all the definitions associated with this program into this namespace

```
namespace game {
   class map { ... };
}
```

- Then these declarations declare objects of different classes:

```
std::map a_std_map; // A standard map
game::map a_game_map;
```

Declaring a Namespace (cont.)

SYNTAX **Namespace Declaration**

FORM:

```
namespace name { ... }
```

EXAMPLE:

```
namespace game { ... }
```

INTERPRETATION:

All identifiers (e.g., functions, variables, etc.) defined within the block following the *name* are within that namespace. Namespaces are open; this means that you may have more than one namespace declaration with the same name in your program. The identifiers defined within the different declarations are merged.

The Global Namespace

- Namespaces are nested:
  - The global namespace is at the top level; its name is effectively the null string
  - By default symbols exist in a global namespace unless they are defined inside a block that starts with keyword `namespace` or members of a class or local variables of a function
  - In our map example, the distinction is between `std::map` and `::map`
  - Since the reference to `::map` is made within the default namespace, the leading prefix `::` (the scope resolution operator) is not required
- If we defined the namespace `game` but did not define the `map` class inside of it, code within the scope of namespace `game` that referenced class `map`, defined in the global namespace, would still require the `::` qualifier, e.g., `::map`

The Global Namespace (cont.)

- The fragment below declares a class `map` in the global namespace and one in the `game` namespace

```
class map { ... }; // Defined in the global namespace
namespace game {
    class map { ... }; // Defined in the game namespace
}
```

- The statement

```
map map1;
```

  in the global namespace declares an object `map1` of type `::map` (that is, the `map` defined in the global namespace)

- The statement

```
game::map map2
```

  in the global namespace declares an object `map2` of type `game::map` (that is the `map` defined in namespace `game`)

The Global Namespace (cont.)

- The statements

```
namespace game {
  map map4; // map4 is the map defined in the game namespace.
  ::map map5; // map5 is the map defined in the global namespace.
}
```

  are in the `game` namespace.
- The object `map4` is of the `map` class that is declared in the `game` namespace
- The object `map5` is of the type `map` defined in the global namespace

The `using` Declaration and `using` Directive

- The `using` *declaration* takes a name from the namespace in which it was declared and places it into the namespace where the `using` declaration appears
  - EXAMPLE:

```
using std::cout;
```

- The qualified name (`cout`) is now a member of the namespace in which the `using` declaration appears

The `using` Declaration and `using` Directive (cont.)

- The `using` *directive* takes all of the names from a given namespace and places them into the namespace where the `using` directive appears
  - EXAMPLE:

```
using namespace std;
```

- All of the names defines in `std` now are defined as such within the current namespace

Using Visibility to Support Encapsulation

Visibility	Applied to Class Members
`private`	Visible only within this class
`protected`	Visible to classes that extend this class
`public`	Visible to all classes and functions

The `friend` Declaration

- The `friend` declaration allows a function external to a class to access the private members of that class
- The `friend` declaration gives the functions and classes it specifies access to the private and protected members of the class in which the `friend` declaration appears
- This effectively makes the named function or class a member of the declaring class
- The class itself must declare who its friends are
- Also, friendship is not inherited and is not transitive
  - This means that if a base class is a friend of a particular class, its derived classes are not automatically friends too
  - They must be declared to be friends

The `friend` Declaration (cont.)

```
class A {
  private:
    int x;
  friend foo;
}

class B {
  void foo() {
    A a;
    // . . .
    a.x = 10; // Valid, foo is a friend of A and a is an object of class A.
  }
}

Class C {
  void bar() {
    A a;
    // . . .
    a.x = 10; // Not valid, x is not visible to bar.
  }
}
```

The `friend` Declaration (cont.)

```
class A {
     private:
          int x;
     friend class B;
};
class B {
     private:
          int y;
          void foo() {
               A a;
               a.x = 10; // Legal, B is a friend of A.
          }
     friend class C;
};
class C {
     void bar() {
          A a;
          B b;
          b.y = 10; // Legal, C is a friend of B.
          a.x = 10; // Not valid, C is not a friend of A.
     }
};
```

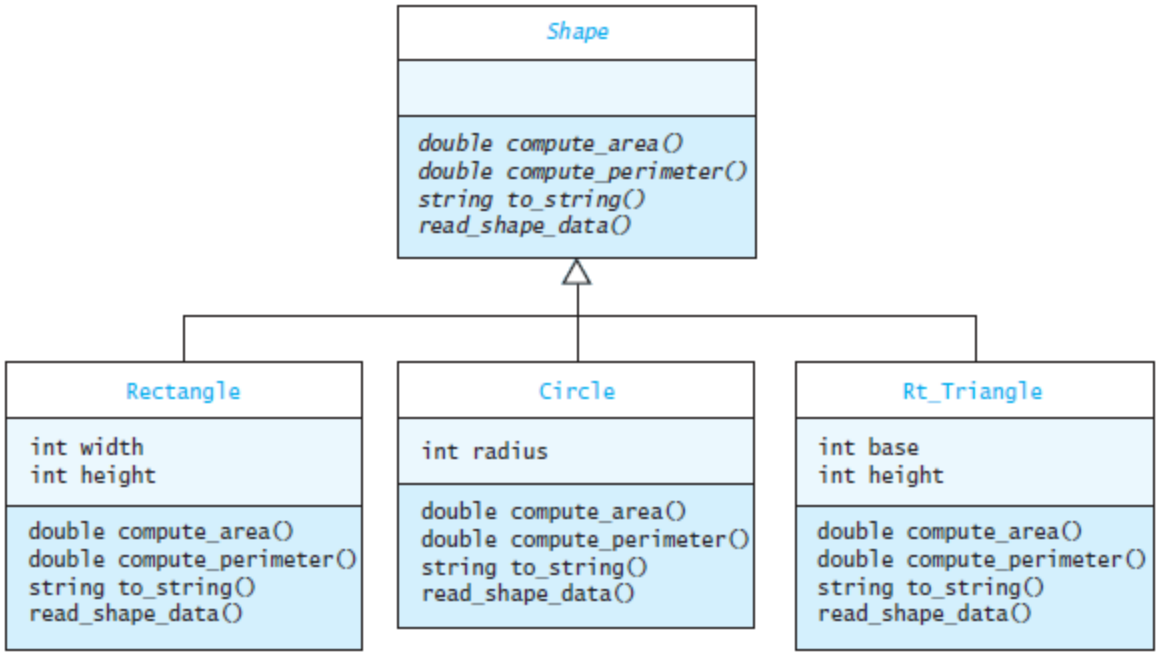
A Shape Class Hierarchy

Section 3.6

Case Study: Processing Geometric Figures

- **PROBLEM**
  - We want to process some standard geometric shapes. Each figure object will be one of three standard shapes (rectangle, circle, right triangle)
  - We want to do standard computations, such as finding the area and perimeter, for any of these shapes
- **ANALYSIS**
  - For each geometric shape, we need a class that represents the shape and knows how to perform the standard computations on it
  - These classes will be Rectangle, Circle, and Rt_Triangle
  - To ensure that these shape classes all define the required computational methods we will require them to implement an abstract class, Shape
  - If a class does not have the required functions, we will get a syntax error when we attempt to use it

Shape **Class Hierarchy**



Class Rectangle

Data Field	Attribute
`int width`	Width of a rectangle.
`int height`	Height of a rectangle.
Function	**Behavior**
`double compute_area() const`	Computes the rectangle area (width × height).
`double compute_perimeter() const`	Computes the rectangle perimeter (2 × width + 2 × height).
`void read_shape_data()`	Reads the width and height.
`string to_string() const`	Returns a string representing the state.

Implementation

```
#ifndef SHAPE_H_
#define SHAPE_H_

#include <string>

/** Declaration file for the abstract class Shape. */
class Shape {
  public:
    virtual double compute_area() const = 0;
    virtual double compute_perimeter() const = 0;
    virtual void read_shape_data() = 0;
    virtual std::string to_string() const = 0;
};

#endif
```

Implementation (cont.)

```
#ifndef RECTANGLE_H_
#define RECTANGLE_H_

#include "Shape.h"
/** Definition file for the class Rectangle. */
class Rectangle : public virtual Shape {
  public:
    Rectangle() : width(0), height(0) {}
    Rectangle(int the_width, int the_height) :
      width(the_width), height(the_height) {}
    int get_width() { return width; }
    int get_height() { return height; }
    double compute_area() const;
    double compute_perimeter() const;
    void read_shape_data();
    std::string to_string() const;
  protected:
    int width;
    int height;
};
```

Implementation (cont.)

```
/** Implementation of Rectangle class. */
#include "Rectangle.h"

#include <sstream>
#include <iostream>
#include <istream>
#include <ostream>

using std::string;
using std::ostringstream;
using std::cin;
using std::cout;

double Rectangle::compute_area() const {
  return width * height;
}

double Rectangle::compute_perimeter() const {
  return 2 * width + 2 * height;
}

void Rectangle::read_shape_data() {
  cout << "Enter the width of the rectangle: ";
  cin >> width;
  cout << "Enter the height of the rectangle: ";
  cin >> height;
}

string Rectangle::to_string() const {
  ostringstream result;
  result << "Rectangle: width is " << width
         << ", height is " << height;
  return result.str();
}
```

Testing

```
/** Program to compute the area and perimeter of geometric shapes. */

#include "Rectangle.h"
#include "Rt_Triangle.h"
#include "Circle.h"

#include <iostream>
#include <istream>
#include <ostream>
#include <cctype>

using std::cout;
using std::cin;
using std::endl;
Shape* get_shape() {
  char fig_type;
  cout << "Enter C for circle\nEnter R for rectangle"
       << "\nEnter T for right triangle\n";
  cin >> fig_type;
  fig_type = tolower(fig_type);
  if (fig_type == 'c') {
    return new Circle();
  } else if (fig_type == 't') {
    return new Rt_Triangle();
  } else if (fig_type == 'r') {
    return new Rectangle();
  } else {
    return NULL;
  }
}

void display_result(double area, double perim) {
  cout << "The area is " << area
       << "\nThe perimeter is " << perim
       << endl;
}

int main() {
  Shape* my_shape;
  double perimeter;
  double area;
  my_shape = get_shape();
  my_shape->read_shape_data();
  perimeter = my_shape->compute_perimeter();
  area = my_shape->compute_area();
  display_result(area, perimeter);
  delete my_shape;
  return 0;
}
```

Testing (cont.)

- The function `get_shape` is an example of a *factory function*
  - it creates a new object and returns a pointer to it
- The author of the `main` function has no awareness of the individual kinds of shapes
- Knowledge of the available shapes is confined to the `get_shape` function; this function must
  - present a list of available shapes to the user
  - decode the user's response
  - return an instance of the desired shape