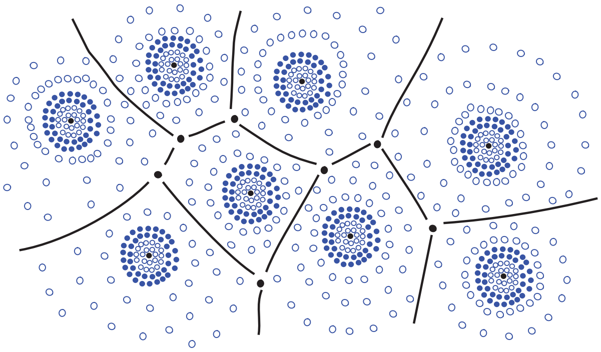
Chapter 2 Object-Oriented Design



Contents

2.1 Goals, Principles, and Patterns

As the name implies, the main "actors" in the object-oriented design paradigm are called ***objects***. An object comes from a ***class***, which is a specification of the data ***members*** that the object contains, as well as the ***member functions*** (also called ***methods*** or operations) that the object can execute. Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects. This view of computing is intended to fulfill several goals and incorporate several design principles, which we discuss in this chapter.

2.1.1 Object-Oriented Design Goals

Software implementations should achieve ***robustness***, ***adaptability***, and ***reusability***. (See Figure 2.1.)

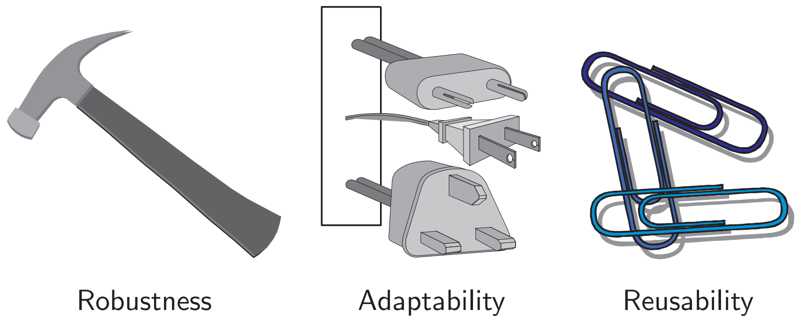


Figure 2.1: Goals of object-oriented design.

Robustness

Every good programmer wants to develop software that is correct, which means that a program produces the right output for all the anticipated inputs in the program's application. In addition, we want software to be ***robust***, that is, capable of handling unexpected inputs that are not explicitly defined for its application. For example, if a program is expecting a positive integer (for example, representing the price of an item) and instead is given a negative integer, then the program should be able to recover gracefully from this error. More importantly, in ***life-critical applications***, where a software error can lead to injury or loss of life, software that is not robust could be deadly. This point was driven home in the late 1980s in accidents involving Therac-25, a radiation-therapy machine, which severely overdosed six patients between 1985 and 1987, some of whom died from complications resulting from their radiation overdose. All six accidents were traced to software errors.

Adaptability

Modern software applications, such as Web browsers and Internet search engines, typically involve large programs that are used for many years. Software therefore needs to be able to evolve over time in response to changing conditions in its environment. Thus, another important goal of quality software is that it achieves ***adaptability*** (also called ***evolvability***). Related to this concept is ***portability***, which is the ability of software to run with minimal change on different hardware and operating system platforms. An advantage of writing software in C++ is the portability provided by the language itself.

Reusability

Going hand in hand with adaptability is the desire that software be reusable, that is, the same code should be usable as a component of different systems in various applications. Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications. Such reuse should be done with care, however, for one of the major sources of software errors in the Therac-25 came from inappropriate reuse of Therac-20 software (which was not object-oriented and not designed for the hardware platform used with the Therac-25).

2.1.2 Object-Oriented Design Principles

Chief among the principles of the object-oriented approach, which are intended to facilitate the goals outlined above, are the following (see Figure 2.2):

- Abstraction
- Encapsulation
- Modularity.

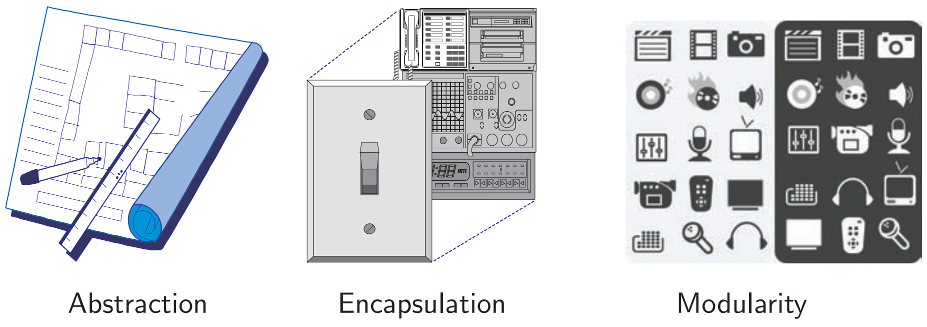


Figure 2.2: Principles of object-oriented design.

Abstraction

The notion of ***abstraction*** is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to ***abstract data types*** (ADTs). An ADT is a mathematical model of a data structure that specifies the type of the data stored, the operations supported on them, and the types of the parameters of the operations. An ADT specifies ***what*** each operation does, but not ***how*** it does it. In C++, the functionality of a data structure is expressed through the public interface of the associated class or classes that define the data structure. By ***public interface***, we mean the signatures (names, return types, and argument types) of a class's public member functions. This is the only part of the class that can be accessed by a user of the class.

An ADT is realized by a concrete data structure, which is modeled in C++ by a ***class***. A class defines the data being stored and the operations supported by the objects that are instances of the class. Also, unlike interfaces, classes specify ***how*** the operations are performed in the body of each function. A C++ class is said to ***implement an interface*** if its functions include all the functions declared in the interface, thus providing a body for them. However, a class can have more functions than those of the interface.

Encapsulation

Another important principle of object-oriented design is the concept of ***encapsulation***, which states that different components of a software system should not reveal the internal details of their respective implementations. One of the main advantages of encapsulation is that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

Modularity

In addition to abstraction and encapsulation, a fundamental principle of object-oriented design is ***modularity***. Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. In object-oriented design, this code structuring approach centers around the concept of ***modularity***. Modularity refers to an organizing principle for code in which different components of a software system are divided into separate functional units.

Hierarchical Organization

The structure imposed by modularity helps to enable software reusability. If software modules are written in an abstract way to solve general problems, then modules can be reused when instances of these same general problems arise in other contexts.

For example, the structural definition of a wall is the same from house to house, typically being defined in terms of vertical studs, spaced at fixed-distance intervals, etc. Thus, an organized architect can reuse his or her wall definitions from one house to another. In reusing such a definition, some parts may require redefinition, for example, a wall in a commercial building may be similar to that of a house, but the electrical system and stud material might be different.

A natural way to organize various structural components of a software package is in a ***hierarchical*** fashion, which groups similar abstract definitions together in a level-by-level manner that goes from specific to more general as one traverses up the hierarchy. A common use of such hierarchies is in an organizational chart where each link going up can be read as "is a," as in "a ranch is a house is a building." This kind of hierarchy is useful in software design, for it groups together common functionality at the most general level, and views specialized behavior as an extension of the general one.

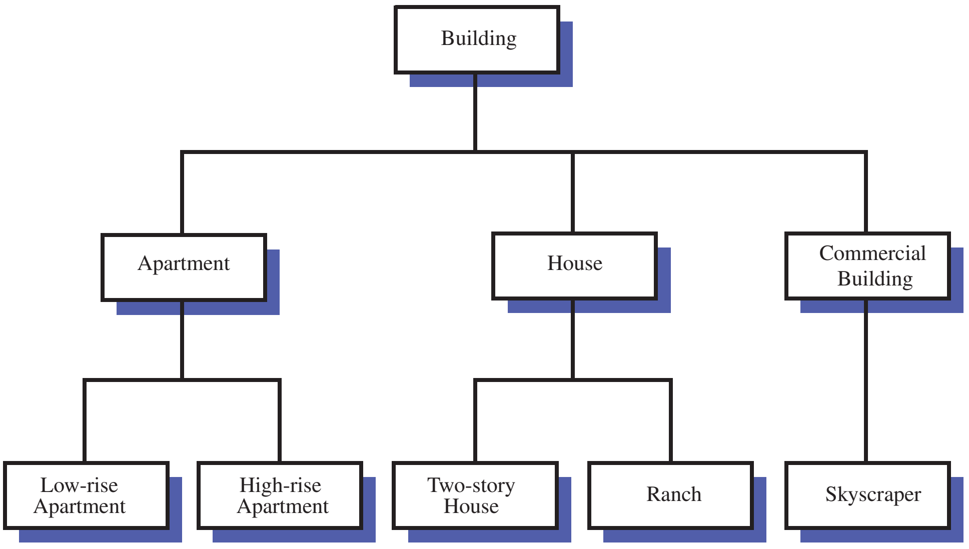


Figure 2.3: An example of an "is a" hierarchy involving architectural buildings.

2.1.3 Design Patterns

One of the advantages of object-oriented design is that it facilitates reusable, robust, and adaptable software. Designing good code takes more than simply understanding object-oriented methodologies, however. It requires the effective use of object-oriented design techniques.

Computing researchers and practitioners have developed a variety of organizational concepts and methodologies for designing quality object-oriented software that is concise, correct, and reusable. Of special relevance to this book is the concept of a ***design pattern***, which describes a solution to a "typical" software design problem. A pattern provides a general template for a solution that can be applied in many different situations. It describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand. It consists of a name, which identifies the pattern, a context, which describes the scenarios for which this pattern can be applied, a template, which describes how the pattern is applied, and a result, which describes and analyzes what the pattern produces.

We present several design patterns in this book, and we show how they can be consistently applied to implementations of data structures and algorithms. These design patterns fall into two groups—patterns for solving algorithm design problems and patterns for solving software engineering problems. Some of the algorithm design patterns we discuss include the following:

- Recursion (Section 3.5)
- Amortization (Section 6.1.3)
- Divide-and-conquer (Section 11.1.1)
- Prune-and-search, also known as decrease-and-conquer (Section 11.5.1)
- Brute force (Section 12.3.1)
- The greedy method (Section 12.4.2)
- Dynamic programming (Section 12.2)

Likewise, some of the software engineering design patterns we discuss include:

- Position (Section 6.2.1)
- Adapter (Section 5.3.4)
- Iterator (Section 6.2.1)
- Template method (Sections 7.3.7, 11.4, and 13.3.3)
- Composition (Section 8.1.2)
- Comparator (Section 8.1.2)
- Decorator (Section 13.3.1)

Rather than explain each of these concepts here, however, we introduce them throughout the text as noted above. For each pattern, be it for algorithm engineering or software engineering, we explain its general use and we illustrate it with at least one concrete example.

2.2 Inheritance and Polymorphism

To take advantage of hierarchical relationships, which are common in software projects, the object-oriented design approach provides ways of reusing code.

2.2.1 Inheritance in C++

The object-oriented paradigm provides a modular and hierarchical organizing structure for reusing code through a technique called ***inheritance***. This technique allows the design of generic classes that can be specialized to more particular classes, with the specialized classes reusing the code from the generic class. For example, suppose that we are designing a set of classes to represent people at a university. We might have a generic class Person, which defines elements common to all people. We could then define specialized classes such as Student, Administrator, and Instructor, each of which provides specific information about a particular type of person.

A generic class is also known as a ***base class***, ***parent class***, or ***superclass***. It defines "generic" members that apply in a multitude of situations. Any class that ***specializes*** or ***extends*** a base class need not give new implementations for the general functions, for it ***inherits*** them. It should only define those functions that are specialized for this particular class. Such a class is called a ***derived class***, ***child class***, or ***subclass***.

Let us consider an example to illustrate these concepts. Suppose that we are writing a program to deal with people at a university. Below we show a partial implementation of a generic class for a person. We use "// ..." to indicate code that is irrelevant to the example and so has been omitted.

```
class Person {                          // Person (base class)
private:
  string         name;                  // name
  string         idNum;                 // university ID number
public:
  // ...
  void print();                         // print information
  string getName();                     // retrieve name
};
```

Suppose we next wish to define a student object. We can derive our class Student from class Person as shown below.

```
class Student : public Person {         // Student (derived from Person)
private:
  string        major;                  // major subject
  int           gradYear;               // graduation year
public:
  // ...
  void print();                         // print information
  void changeMajor(const string& newMajor); // change major
};
```

The "public Person" phrase indicates that the Student is derived from the Person class. (The keyword "**public**" specifies ***public inheritance***. We discuss other types of inheritance later.) When we derive classes in this way, there is an implied "is a" relationship between them. In this case, a Student "is a" Person. In particular, a Student object inherits all the member data and member functions of class Person in addition to providing its own members. The relationship between these two classes is shown graphically in a ***class inheritance diagram*** in Figure 2.4.

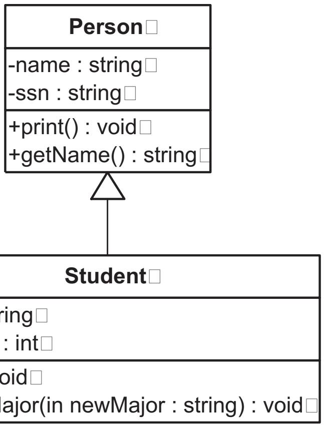


Figure 2.4: A class inheritance diagram, showing a base class Person and derived class Student. Entries tagged with "–" are private and entries tagged with "+" are public. Each block of the diagram consists of three parts: the class name, the class member variables, and the class member functions. The type (or return type) of each member is indicated after the colon (":"). The arrow indicates that Student is derived from Person.

Member Functions

An object of type Person can access the public members of Person. An object of type Student can access the public members of both classes. If a Student object invokes the shared print function, it will use its own version by default. We use the ***class scope operator*** (::) to specify which class's function is used, as in Person::print and Student::print. Note that an object of type Person cannot access members of the base type, and thus it is not possible for a Person object to invoke the changeMajor function of class Student.

```
Person person("Mary", "12-345");    // declare a Person
Student student("Bob", "98-764", "Math", 2012); // declare a Student

cout << student.getName() << endl; // invokes Person::getName()
person.print();                     // invokes Person::print()
student.print();                    // invokes Student::print()
person.changeMajor("Physics");      // ERROR!
student.changeMajor("English");     // okay
```

C++ programmers often find it useful for a derived class to explicitly invoke a member function of a base class. For example, in the process of printing information for a student, it is natural to first print the information of the Person base class, and then print information particular to the student. Performing this task is done using the class scope operator.

```
void Person::print() {              // definition of Person print
  cout << "Name " << name << endl;
  cout << "IDnum " << idNum << endl;
}

void Student::print() {             // definition of Student print
  Person::print();                  // first print Person information
  cout << "Major " << major << endl;
  cout << "Year " << gradYear << endl;
}
```

Without the "Person::" specifier used above, the Student::print function would call itself recursively, which is not what we want.

Protected Members

Even though class Student is inherited from class Person, member functions of Student do not have access to private members of Person. For example, the following is illegal.

```
void Student::printName() {
  cout << name << '\n';          // ERROR! name is private to Person
}
```

Special access privileges for derived classes can be provided by declaring members to be "**protected**." A protected member is "public" to all classes derived from this one, but "private" to all other functions. From a syntactic perspective, the keyword **protected** behaves in the same way as the keyword **private** and **public**. In the class example above, had we declared *name* to be protected rather than private, the above function printName would work fine.

Although C++ makes no requirements on the order in which the various sections of a class appear, there are two common ways of doing it. The first is to declare public members first and private members last. This emphasizes the elements that are important to a user of the class. The other way is to present private members first and public members last. This tends to be easier to read for an implementor. Of course, clarity is a more important consideration than adherence to any standard.

Illustrating Class Protection

Consider for example, three classes: a base class Base, a derived class Derived, and an unrelated class Unrelated. The base class defines three integer members, one of each access type.

```
class Base {
  private:     int priv;
  protected:   int prot;
  public:      int publ;
};

class Derived: public Base {
  void someMemberFunction() {
    cout << priv;                  // ERROR: private member
    cout << prot;                  // okay
    cout << publ;                  // okay
  }
};

class Unrelated {
  Base X;

  void anotherMemberFunction() {
    cout << X.priv;                // ERROR: private member
    cout << X.prot;                // ERROR: protected member
    cout << X.publ;                // okay
  }
};
```

When designing a class, we should give careful thought to the access privileges we give each member variable or function. Member variables are almost always declared to be private or at least protected, since they determine the details of the class's implementation. A user of the class can access only the public class members, which consist of the principal member functions for accessing and manipulating class objects. Finally, protected members are commonly used for utility functions, which may be useful to derived classes. We will see many examples of these three access types in the examples appearing in later chapters.

Constructors and Destructors

We saw in Section 1.5.2, that when a class object is created, the class's constructor is called. When a derived class is constructed, it is the responsibility of this class's constructor to take care that the appropriate constructor is called for its base class. Class hierarchies in C++ are constructed bottom-up: base class first, then its members, then the derived class itself. For this reason, the constructor for a base class needs to be called in the initializer list (see Section 1.5.2) of the derived class. The example below shows how constructors might be implemented for the Person and Student classes.

```
Person::Person(const string& nm, const string& id)
  : name(nm),                              // initialize name
    idNum(id) { }                          // initialize ID number

Student::Student(const string& nm, const string& id,
                 const string& maj, int year)
  : Person(nm, id),                        // initialize Person members
  major(maj),                              // initialize major
  gradYear(year) { }                       // initialize graduation year
```

Only the Person(nm, id) call has to be in the initializer list. The other initializations could be placed in the constructor function body ({...}), but putting class initializations in the initialization list is generally more efficient. Suppose that we create a new student object.

```
Student* s = new Student("Carol", "34-927", "Physics", 2014);
```

Note that the constructor for the Student class first makes a function call to Person("Carol", "34-927") to initialize the Person base class, and then it initializes the major to "Physics" and the year to 2014.

Classes are destroyed in the reverse order from their construction, with derived classes destroyed before base classes. For example, suppose that we declared destructors for these two classes. (Note that destructors are not really needed in this case, because neither class allocates storage or other resources.)

```
Person::~Person() { ... }                  // Person destructor
Student::~Student() { ... }                // Student destructor
```

If we were to destroy our student object, the Student destructor would be called first, followed by the Person destructor. Unlike constructors, the Student destructor does not need to (and is not allowed to) call the Person destructor. This happens automatically.

```
delete s;                                   // calls ~Student() then ~Person()
```

Static Binding

When a class is derived from a base class, as with Student and Person, the derived class becomes a ***subtype*** of the base class, which means that we can use the derived class wherever the base class is acceptable. For example, suppose that we create an array of pointers to university people.

```
Person* pp[100];                            // array of 100 Person pointers
pp[0] = new Person(...);                    // add a Person (details omitted)
pp[1] = new Student(...);                   // add a Student (details omitted)
```

Since getName is common to both classes, it can be invoked on either elements of the array. A more interesting issue arises if we attempt to invoke print. Since $pp[1]$ holds the address of a Student object, we might think that the function Student::print would be called. Surprisingly, the function Person::print is called in both cases, in spite of the apparent difference in the two objects. Furthermore, $pp[i]$ is not even allowed to access Student member functions.

```
cout << pp[1]->getName() << '\n'; // okay
pp[0]->print();                    // calls Person::print()
pp[1]->print();                    // also calls Person::print() (!)
pp[1]->changeMajor("English");     // ERROR!
```

The reason for this apparently anomalous behavior is called ***static binding***—when determining which member function to call, C++'s default action is to consider an object's ***declared type***, not its actual type. Since $pp[1]$ is declared to be a pointer to a Person, the members for that class are used. Nonetheless, C++ provides a way to achieve the desired dynamic effect using the technique we describe next.

Dynamic Binding and Virtual Functions

As we saw above, C++ uses ***static binding*** by default to determine which member function to call for a derived class. Alternatively, in ***dynamic binding***, an object's contents determine which member function is called. To specify that a member function should use dynamic binding, the keyword "**virtual**" is added to the function's declaration. Let us redefine our Person and Student, but this time we will declare the print function to be virtual.

```
class Person {                          // Person (base class)
  virtual void print() { ... }          // print (details omitted)
  // ...
};
class Student : public Person {         // Student (derived from Person)
  virtual void print() { ... }          // print (details omitted)
  // ...
};
```

Let us consider the effect of this change on our array example, thereby illustrating the usefulness of dynamic binding.

```
Person* pp[100];                        // array of 100 Person pointers
pp[0] = new Person(...);                // add a Person (details omitted)
pp[1] = new Student(...);               // add a Student (details omitted)
pp[0]->print();                         // calls Person::print()
pp[1]->print();                         // calls Student::print()
```

In this case, *pp*[1] contains a pointer to an object of type Student, and by the power of dynamic binding with virtual functions, the function Student::print will be called. The decision as to which function to call is made at run-time, hence the name ***dynamic binding***.

Virtual Destructors

There is no such thing as a virtual constructor. Such a concept does not make any sense. Virtual destructors, however, are very important. In our array example, since we store objects of both types Person and Student in the array, it is important that the appropriate destructor be called for each object. However, if the destructor is nonvirtual, then only the Person destructor will be called in each case. In our example, this choice is not a problem. But if the Student class had allocated memory dynamically, the fact that the wrong destructor is called would result in a memory leak (see Section 1.5.3).

When writing a base class, we cannot know, in general, whether a derived class may need to implement a destructor. So, to be safe, when defining any virtual functions, it is recommended that a virtual destructor be defined as well. This destructor may do nothing at all, and that is fine. It is provided just in case a derived class needs to define its own destructor. This principle is encapsulated in the following rule of thumb.

Remember

If a base class defines any virtual functions, it should define a ***virtual destructor***, even if it is empty.

Dynamic binding is a powerful technique, since it allows us to create an object, such as the array *pp* above, whose behavior varies depending on its contents. This technique is fundamental to the concept of polymorphism, which we discuss in the next section.

2.2.2 Polymorphism

Literally, "polymorphism" means "many forms." In the context of object-oriented design, it refers to the ability of a variable to take different types. Polymorphism is typically applied in C++ using pointer variables. In particular, a variable *p* declared to be a pointer to some class S implies that *p* can point to any object belonging to any derived class T of S.

Now consider what happens if both of these classes define a virtual member function a, and let us consider which of these functions is called when we invoke *p*->a(). Since dynamic binding is used, if *p* points to an object of type T, then it invokes the function T::a. In this case, T is said to ***override*** function a from S. Alternatively, if *p* points to an object of type S, it will invoke S::a.

Polymorphism such as this is useful because the caller of *p*->a() does not have to know whether the pointer *p* refers to an instance of T or S in order to get the a function to execute correctly. A pointer variable *p* that points to a class object that has at least one virtual function is said to be ***polymorphic***. That is, *p* can take many forms, depending on the specific class of the object it is referring to. This kind of functionality allows a specialized class T to extend a class S, inherit the "generic" functions from class S, and redefine other functions from class S to account for specific properties of objects of class T.

Inheritance, polymorphism, and function overloading support reusable software. We can define classes that inherit generic member variables and functions and can then define new, more specific variables and functions that deal with special aspects of objects of the new class. For example, suppose that we defined a generic class Person and then derived three classes Student, Administrator, and Instructor. We could store pointers to all these objects in a list of type Person*. When we invoke a virtual member function, such as print, to any element of the list, it will call the function appropriate to the individual element's type.

Specialization

There are two primary ways of using inheritance, one of which is ***specialization***. In using specialization, we are specializing a general class to a particular derived class. Such derived classes typically possess an "is a" relationship to their base class. The derived classes inherit all the members of the base class. For each inherited function, if that function operates correctly, independent of whether it is operating for a specialization, no additional work is needed. If, on the other hand, a general function of the base class would not work correctly on the derived class, then we should override the function to have the correct functionality for the derived class.

For example, we could have a general class, Dog, which has a function drink and a function sniff. Specializing this class to a Bloodhound class would probably not require that we override the drink function, as all dogs drink pretty much the same way. But it could require that we override the sniff function, as a Bloodhound has a much more sensitive sense of smell than a "generic" dog. In this way, the Bloodhound class specializes the functions of its base class, Dog.

Extension

Another way of using inheritance is ***extension***. In using extension, we reuse the code written for functions of the base class, but we then add new functions that are not present in the base class, so as to extend its functionality. For example, returning to our Dog class, we might wish to create a derived class, BorderCollie, which inherits all the generic functions of the Dog class, but then adds a new function, herd, since Border Collies have a herding instinct that is not present in generic dogs, thereby extending the functionality of a generic dog.

2.2.3 Examples of Inheritance in C++

To make the concepts of inheritance and polymorphism more concrete, let us consider a simple example in C++. We consider an example of several classes that print numeric progressions. A ***numeric progression*** is a sequence of numbers, where the value of each number depends on one or more of the previous values. For example, an ***arithmetic progression*** determines a next number by addition of a fixed increment. A ***geometric progression*** determines a next number by multiplication by a fixed base value. In any case, a progression requires a way of defining its first value and it needs a way of identifying the current value as well.

Arithmetic progression (increment 1)	$0,1,2,3,4,5,\ldots$
Arithmetic progression (increment 3)	$0,3,6,9,12,\ldots$
Geometric progression (base 2)	$1,2,4,8,16,32,\ldots$
Geometric progression (base 3)	$1,3,9,27,81,\ldots$

We begin by defining a class, Progression, which is declared in the code fragment below. It defines the "generic" members and functions of a numeric progression. Specifically, it defines the following two long-integer variable members:

- *first*: first value of the progression
- *cur*: current value of the progression

Because we want these variables to be accessible from derived classes, we declare them to be protected.

We define a constructor, Progression, a destructor, ˜Progression, and the following three member functions.

firstValue():	Reset the progression to the first value and return it.
nextValue():	Step the progression to the next value and return it.
printProgression(n):	Reset the progression and print its first n values.

```
class Progression {                          // a generic progression
public:
  Progression(long f = 0)                    // constructor
    : first(f), cur(f) { }
  virtual ˜Progression() { };                // destructor
  void printProgression(int n);              // print the first n values
protected:
  virtual long firstValue();                 // reset
  virtual long nextValue();                  // advance
protected:
  long first;                                // first value
  long cur;                                  // current value
};
```

The member function printProgression is public and is defined below.

```
void Progression::printProgression(int n) {  // print n values
  cout << firstValue();                      // print the first
  for (int i = 2; i <= n; i++)               // print 2 through n
    cout << ' ' << nextValue();
  cout << endl;
}
```

In contrast, the member functions firstValue and nextValue are intended as utilities that will only be invoked from within this class or its derived classes. For this reason, we declare them to be protected. They are defined below.

```
  long Progression::firstValue() {          // reset
    cur = first;
    return cur;
  }
  long Progression::nextValue() {           // advance
    return ++cur;
  }
```

It is our intention that, in order to generate different progressions, derived classes will override one or both of these functions. For this reason, we have declared both to be virtual. Because there are virtual functions in our class, we have also provided a virtual destructor in order to be safe. (Recall the discussion of virtual destructors from Section 2.2.1.) At this point the destructor does nothing, but this might be overridden by derived classes.

Arithmetic Progression Class

Let us consider a class ArithProgression, shown below. We add a new member variable *inc*, which provides the value to be added to each new element of the progression. We also override the member function nextValue to produce the desired new behavior.

```
class ArithProgression : public Progression {  // arithmetic progression
public:
  ArithProgression(long i = 1);                 // constructor
protected:
  virtual long nextValue();                     // advance
protected:
  long inc;                                     // increment
};
```

The constructor and the new member function nextValue are defined below. Observe that the constructor invokes the base class constructor Progression to initialize the base object in addition to initializing the value of *inc*.

```
  ArithProgression::ArithProgression(long i)    // constructor
    : Progression(), inc(i) { }

  long ArithProgression::nextValue() {          // advance by adding
    cur += inc;
    return cur;
  }
```

Polymorphism is at work here. When a Progression pointer is pointing to an ArithProgression object, it will use the ArithProgression functions firstValue and nextValue. Even though the function printProgression is not virtual, it makes use of this polymorphism. Its calls to the firstValue and nextValue functions are implicitly for the "current" object, which will be of the ArithProgression class.

A Geometric Progression Class

Let us next define GeomProgression that implements a geometric progression. As with the ArithProgression class, this new class inherits the member variables *first* and *cur*, and the member functions firstValue and printProgression from Progression. We add a new member variable *base*, which holds the base value to be multiplied to form each new element of the progression. The constructor initializes the base class with a starting value of 1 rather than 0. The function nextValue applies multiplication to obtain the next value.

```
class GeomProgression : public Progression { // geometric progression
public:
  GeomProgression(long b = 2);                // constructor
protected:
  virtual long nextValue();                   // advance
protected:
  long base;                                  // base value
};

GeomProgression::GeomProgression(long b)      // constructor
  : Progression(1), base(b) { }

long GeomProgression::nextValue() {           // advance by multiplying
  cur *= base;
  return cur;
}
```

A Fibonacci Progression Class

As a further example, we define a FibonacciProgression class that represents another kind of progression, the ***Fibonacci progression***, where the next value is defined as the sum of the current and previous values. We show the FibonacciProgression class below. Recall that each element of a Fibonacci series is the sum of the previous two elements.

$$\text{Fibonacci progression (first} = 0\text{, second} = 1\text{):}\quad 0,1,1,2,3,5,8,\ldots$$

In addition to the current value *cur* in the Progression base class, we also store here the value of the previous element, denoted *prev*. The constructor is given the first two elements of the sequence. The member variable *first* is inherited from the base class. We add a new member variable *second*, to store this second element. The default values for the first and second elements are 0 and 1, respectively.

```
class FibonacciProgression : public Progression { // Fibonacci progression
public:
  FibonacciProgression(long f = 0, long s = 1); // constructor
protected:
  virtual long firstValue();                  // reset
  virtual long nextValue();                   // advance
protected:
  long second;                                // second value
  long prev;                                  // previous value
};
```

The initialization process is a bit tricky because we need to create a “fictitious” element that precedes the first element. Note that setting this element to the value *second − first* achieves the desired result. This change is reflected both in the constructor and the overridden member function firstValue. The overridden member function nextValue copies the current value to the previous value. We need to store the old previous value in a temporary variable.

```
FibonacciProgression::FibonacciProgression(long f, long s)
  : Progression(f), second(s), prev(second − first)  { }

long FibonacciProgression::firstValue() {          // reset
  cur = first;
  prev = second − first;                           // create fictitious prev
  return cur;
}

long FibonacciProgression::nextValue() {           // advance
  long temp = prev;
  prev = cur;
  cur += temp;
  return cur;
}
```

Combining the Progression Classes

In order to visualize how the three different progression classes are derived from the generic Progression class, we give their inheritance diagram in Figure 2.5.

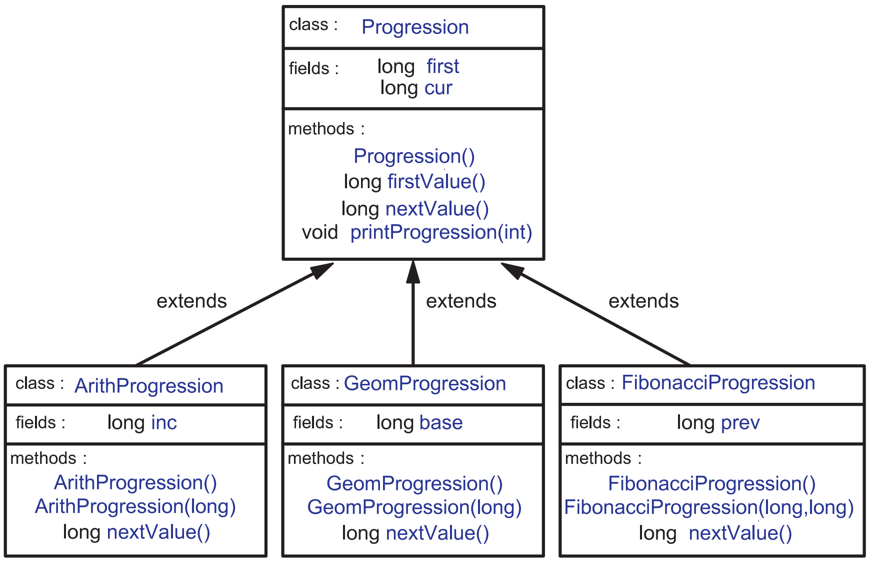


Figure 2.5: Inheritance diagram for class Progression and its subclasses.

To complete our example, we define the main function shown in Code Fragment 2.1, which performs a simple test of each of the three classes. In this class, variable *prog* is a polymorphic array of pointers to class Progression. Since each of its members points to an object of class ArithProgression, GeomProgression, or FibonacciProgression, the functions appropriate to the given progression are invoked in each case. The output is shown in Code Fragment 2.2. Notice that this program has a (unimportant) memory leak because we never deleted the allocated object.

The example presented in this section provides a simple illustration of inheritance and polymorphism in C++. The Progression class, its derived classes, and the tester program have a number of shortcomings, however, which might not be immediately apparent. One problem is that the geometric and Fibonacci progressions grow quickly, and there is no provision for handling the inevitable overflow of the long integers involved. For example, since $3^{40} > 2^{63}$, a geometric progression with base $b = 3$ will overflow a 64-bit long integer after 40 iterations. Likewise, the 94th Fibonacci number is greater than 2^{63}; hence, the Fibonacci progression will overflow a 64-bit long integer after 94 iterations. Another problem is that we may not allow arbitrary starting values for a Fibonacci progression. For example, do we allow a Fibonacci progression starting with 0 and -1? Dealing with input errors or error conditions that occur during the running of a C++ program requires that we have some mechanism for handling them. We discuss this topic later in Section 2.4.

2.2.4 Multiple Inheritance and Class Casting

In the examples we have shown so far, a subclass has been derived from a single base class and we didn't have to deal with the problem of viewing an object of a specific declared class as also being of an inherited type. We discuss some related, more-advanced C++ programming issues in this section.

Multiple and Restricted Inheritance

In C++, we are allowed to derive a class from a number of base classes, that is, C++ allows ***multiple inheritance***. Although multiple inheritance can be useful, especially in defining interfaces, it introduces a number of complexities. For example, if both base classes provide a member variable with the same name or a member function with the same declaration, the derived class must specify from which base class the member should be used (which is complicated). For this reason, we use single inheritance almost exclusively.

We have been using public inheritance in our previous examples, indicated by the keyword **public** in specifying the base class. Remember that private base class members are not accessible in a derived class. Protected and public members of the base class become protected and public members of the derived class, respectively.

```
/** Test program for the progression classes */
int main() {
  Progression* prog;
                                        // test ArithProgression
  cout << "Arithmetic progression with default increment:\n";
  prog = new ArithProgression();
  prog−>printProgression(10);
  cout << "Arithmetic progression with increment 5:\n";
  prog = new ArithProgression(5);
  prog−>printProgression(10);
                                        // test GeomProgression
  cout << "Geometric progression with default base:\n";
  prog = new GeomProgression();
  prog−>printProgression(10);
  cout << "Geometric progression with base 3:\n";
  prog = new GeomProgression(3);
  prog−>printProgression(10);
                                        // test FibonacciProgression
  cout << "Fibonacci progression with default start values:\n";
  prog = new FibonacciProgression();
  prog−>printProgression(10);
  cout << "Fibonacci progression with start values 4 and 6:\n";
  prog = new FibonacciProgression(4, 6);
  prog−>printProgression(10);
  return EXIT_SUCCESS;                  // successful execution
}
```

Code Fragment 2.1: Program for testing the progression classes.

```
Arithmetic progression with default increment:
0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5:
0 5 10 15 20 25 30 35 40 45
Geometric progression with default base:
1 2 4 8 16 32 64 128 256 512
Geometric progression with base 3:
1 3 9 27 81 243 729 2187 6561 19683
Fibonacci progression with default start values:
0 1 1 2 3 5 8 13 21 34
Fibonacci progression with start values 4 and 6:
4 6 10 16 26 42 68 110 178 288
```

Code Fragment 2.2: Output of TestProgression program from Code Fragment 2.1.

C++ supports two other types of inheritance. These different types of inheritance diminish the access rights for base class members. In ***protected inheritance***, fields declared to be public in the base class become protected in the child class. In ***private inheritance***, fields declared to be public and protected in the base class become private in the derived class. An example is shown below.

```
class Base {                              // base class
  protected:    int foo;
  public:       int bar;
};

class Derive1 : public Base {             // public inheritance
  // foo is protected and bar is public
};

class Derive2 : protected Base {          // protected inheritance
  // both foo and bar are protected
};

class Derive3 : private Base {            // public inheritance
  // both foo and bar are private
};
```

Protected and private inheritance are not used as often as public inheritance. We only use public inheritance in this book.

Casting in an Inheritance Hierarchy

An object variable can be viewed as being of various types, but it can be declared as only one type. Thus, a variable's declared type determines how it is used, and even determines how certain functions will act on it. Enforcing that all variables be typed and that operations declare the types they expect is called ***strong typing***, which helps prevent bugs. Nonetheless, we sometimes need to explicitly change, or ***cast***, a variable from one type to another. We have already introduced type casting in Section 1.2.1. We now discuss how it works for classes.

To illustrate an example where we may want to perform a cast, recall our class hierarchy consisting of a base class Person and derived class Student. Suppose that we are storing pointers to objects of both types in an array *pp*. The following attempt to change a student's major would be flagged as an error by the compiler.

```
Person* pp[100];                          // array of 100 Person pointers
pp[0] = new Person(...);                  // add a Person (details omitted)
pp[1] = new Student(...);                 // add a Student (details omitted)
// ...
pp[1]−>changeMajor("English");            // ERROR!
```

The problem is that the base class Person does not have a function changeMajor. Notice that this is different from the case of the function print because the print function was provided in both classes. Nonetheless, we "know" that *pp*[1] points to an object of class Student, so this operation should be legal.

To access the changeMajor function, we need to cast the *pp*[1] pointer from type Person* to type Student*. Because the contents of a variable are dynamic, we need to use the C++ run-time system to determine whether this cast is legal, which is what a ***dynamic cast*** does. The syntax of a dynamic cast is shown below.

dynamic_cast < *desired_type* > (*expression*)

Dynamic casting can only be applied to polymorphic objects, that is, objects that come from a class with at least one virtual function. Below we show how to use dynamic casting to change the major of *pp*[1].

```
Student* sp = dynamic_cast<Student*>(pp[1]); // cast pp[1] to Student*
sp−>changeMajor("Chemistry");               // now changeMajor is legal
```

Dynamic casting is most often applied for casting pointers within the class hierarchy. If an illegal pointer cast is attempted, then the result is a null pointer. For example, we would get a NULL pointer from an attempt to cast *pp*[0] as above, since it points to a Person object.

To illustrate the use of dynamic cast, we access all the elements of the *pp* array and, for objects of (actual) type Student, change the major to "Undecided"

```
for (int i = 0; i < 100; i++) {
  Student *sp = dynamic_cast<Student*>(pp[i]);
  if (sp != NULL)                            // cast succeeded?
    sp−>changeMajor("Undecided");            // change major
}
```

The casting we have discussed here could also have been done using the traditional C-style cast or through a static cast (recall Section 1.2.1). Unfortunately, no error checking would be performed in that case. An attempt to cast a Person object pointer to a Student pointer would succeed "silently," but any attempt to use such a pointer would have disastrous consequences.

2.2.5 Interfaces and Abstract Classes

For two objects to interact, they must "know" about each other's member functions. To enforce this "knowledge," the object-oriented design paradigm asks that classes specify the ***application programming interface*** (API), or simply ***interface***, that their objects present to other objects. In the ***ADT-based*** approach (see Section 2.1.2) to data structures followed in this book, an interface defining an ADT is specified as a type definition and a collection of member functions for this type, with the arguments for each function being of specified types.

Some programming languages provide a mechanism for defining ADTs. One example is Java's ***interface***. An interface is a collection of function declarations with no data and no bodies. That is, the member functions of an interface are always empty. When a class implements an interface, it must implement all of the member functions declared in the interface.

C++ does not provide a direct mechanism for specifying interfaces. Nonetheless, throughout this book we often provide ***informal interfaces***, even though they are not legal C++ structures. For example, a ***stack*** data structure (see Chapter 5) is a container that supports various operations such as inserting (or ***pushing***) an element onto the top of the stack, removing (or ***popping***) an element from the top of the stack, and testing whether the stack is empty. Below we provide an example of a minimal interface for a stack of integers.

```
class Stack {                               // informal interface – not a class
public:
  bool isEmpty() const;                     // is the stack empty?
  void push(int x);                         // push x onto the stack
  int pop();                                // pop the stack and return result
};
```

Abstract Classes

The above informal interface is ***not*** a valid construct in C++; it is just a documentation aid. In particular, it does not contain any data members or definitions of member functions. Nonetheless, it is useful, since it provides important information about a stack's public member functions and how they are called.

An ***abstract class*** in C++ is a class that is used only as a base class for inheritance; it cannot be used to create instances directly. At first the idea of creating a class that cannot be instantiated seems to be nonsense, but it is often very important. For example, suppose that we want to define a set of geometric shape classes, say, Circle, Rectangle, and Triangle. It is natural to derive these related classes from a single generic base class, say, Shape. Each of the derived classes will have a virtual member function draw, which draws the associated object. The rules of inheritance require that we define such a function for the base class, but it is unclear what such a function means for a generic shape.

One way to handle this would be to define Shape::draw with an empty function body ({ }), which would be a rather unnatural solution. What is really desired here is some way to inform the compiler that the class Shape is ***abstract***; it is not possible to create objects of type Shape, only its subclasses. In C++, we define a class as being abstract by specifying that one or more members of its functions are ***abstract***, or ***pure virtual***. A function is declared pure virtual by giving "=0" in place of its body. C++ does not allow the creation of an object that has one or more pure virtual functions. Thus, any derived class must provide concrete definitions for all pure virtual functions of the base class.

As an example, recall our Progression class and consider the member function nextValue, which computes the next value in the progression. The meaning of this function is clear for each of the derived classes: ArithProgression, GeomProgression, and FibonacciProgression. However, in the base class Progression we invented a rather arbitrary default for the nextValue function. (Go back and check it. What progression does it compute?) It would be more natural to leave this function undefined. We show below how to make it a ***pure virtual*** member function.

```
class Progression {                         // abstract base class
  // ...
  virtual long nextValue() = 0;             // pure virtual function
  // ...
};
```

As a result, the compiler will not allow the creation of objects of type Progression, since the function nextValue is "pure virtual." However, its derived classes, ArithProgression for example, can be defined because they provide a definition for this member function.

Interfaces and Abstract Base Classes

We said above that C++ does not provide a direct mechanism for defining interfaces for abstract data types. Nevertheless, we can use abstract base classes to achieve much of the same purpose.

In particular, we may construct a class for an interface in which all the functions are pure virtual as shown below for the example of a simple stack ADT.

```
class Stack {                              // stack interface as an abstract class
public:
  virtual bool isEmpty() const = 0;        // is the stack empty?
  virtual void push(int x) = 0;            // push x onto the stack
  virtual int pop() = 0;                   // pop the stack and return result
};
```

A class that implements this stack interface can be derived from this abstract base class, and then provide concrete definitions for all of these virtual functions as shown below.

```
class ConcreteStack : public Stack {  // implements Stack
public:
  virtual bool isEmpty() { ... }       // implementation of members
  virtual void push(int x) { ... }     // ... (details omitted)
  virtual int pop() { ... }
private:
  // ...                               // member data for the implementation
};
```

There are practical limitations to this method of defining interfaces, so we only use informal interfaces for the purpose of illustrating ADTs.

2.3 Templates

Inheritance is only one mechanism that C++ provides in support of polymorphism. In this section, we consider another way—using ***templates***.

2.3.1 Function Templates

Let us consider the following function, which returns the minimum of two integers.

```
int integerMin(int a, int b)           // returns the minimum of a and b
  { return (a < b ? a : b); }
```

Such a function is very handy, so we might like to define a similar function for computing the minimum of two variables of other types, such as long, short, float, and double. Each such function would require a different declaration and definition, however, and making many copies of the same function is an error-prone solution, especially for longer functions.

C++ provides an automatic mechanism, called the ***function template***, to produce a generic function for an arbitrary type T. A function template provides a well-defined pattern from which a concrete function may later be formally defined or ***instantiated***. The example below defines a genericMin function template.

```
template <typename T>
T genericMin(T a, T b) {               // returns the minimum of a and b
  return (a < b ? a : b);
}
```

The declaration takes the form of the keyword "**template**" followed by the notation <typename T>, which is the parameter list for the template. In this case, there is just one parameter T. The keyword “**typename**” indicates that T is the name of some type. (Older versions of C++ do not support this keyword and instead the keyword “**class**” must be used.) We can have other types of template parameters, integers for example, but type names are the most common. Observe that the type parameter T takes the place of “**int**” in the original definition of the genericMin function.

We can now invoke our templated function to compute the minimum of objects of many different types. The compiler looks at the argument types and determines which form of the function to ***instantiate***.

```
cout << genericMin(3, 4) << ' '      // = genericMin<int>(3,4)
  << genericMin(1.1, 3.1) << ' '     // = genericMin<double>(1.1, 3.1)
  << genericMin('t', 'g') << endl;   // = genericMin<char>('t','g')
```

The template type does not need to be a fundamental type. We could use any type in this example, provided that the less than operator (<) is defined for this type.

2.3.2 Class Templates

In addition to function templates, C++ allows classes to be templated, which is a powerful mechanism because it allows us to provide one data structure declaration that can be applied to many different types. In fact, the Standard Template Library uses class templates extensively.

Let us consider an example of a template for a restricted class BasicVector that stores a vector of elements, which is a simplified version of a structure discussed in greater detail in Chapter 6. This class has a constructor that is given the size of the array to allocate. In order to access elements of the array, we overload the indexing operator “[ ].”

We present a partial implementation of a class template for class BasicVector below. We have omitted many of the other member functions, such as the copy constructor, assignment operator, and destructor. The template parameter T takes the place of the actual type that will be stored in the array.

```
template <typename T>
class BasicVector {                    // a simple vector class
public:
  BasicVector(int capac = 10);         // constructor
  T& operator[](int i)                 // access element at index i
    { return a[i]; }
  // ... other public members omitted
private:
  T* a;                                // array storing the elements
  int capacity;                        // length of array a
};
```

We have defined one member function (the indexing operator) within the class body, and below we show how the other member function (the constructor) can be defined outside the class body. The constructor initializes the capacity value and allocates the array storage.

```
template <typename T>                    // constructor
BasicVector<T>::BasicVector(int capac) {
  capacity = capac;
  a = new T[capacity];                   // allocate array storage
}
```

To ***instantiate*** a concrete instance of the class BasicVector, we provide the class name followed by the actual type parameter enclosed in angled brackets (<...>). The code fragment below shows how we would define three vectors, one of type **int**, one of type **double**, and one of type string.

```
BasicVector<int>       iv(5);            // vector of 5 integers
BasicVector<double>  dv(20);             // vector of 20 doubles
BasicVector<string>    sv(10);           // vector of 10 strings
```

Since we have overloaded the indexing operator, we can access elements of each array in the same manner as we would for any C++ array.

```
iv[3] = 8;
dv[14] = 2.5;
sv[7] = "hello";
```

Templated Arguments

The actual argument in the instantiation of a class template can itself be a templated type. For example, we could create a BasicVector whose individual elements are themselves of type BasicVector<int>.

```
BasicVector<BasicVector<int> > xv(5); // a vector of vectors
// ...
xv[2][8] = 15;
```

In this case, because no capacity argument could be provided to the constructor, each element of the vector is constructed using the default capacity of 10. Thus the above definition declares a BasicVector consisting of five elements, each of which is a BasicVector consisting of 10 integers. Such a structure therefore behaves much like a two-dimensional array of integers.

Note that in the declaration of *xv* above, we intentionally left a space after "**<int>**." The reason is that without the space, the character combination ">>" would be interpreted as a bitwise right-shift operator by the compiler (see Section 1.2).

2.4 Exceptions

Exceptions are unexpected events that occur during the execution of a program. An exception can be the result of an error condition or simply an unanticipated input. In C++, exceptions can be thought of as being objects themselves.

2.4.1 Exception Objects

In C++, an exception is "***thrown***" by code that encounters some unexpected condition. Exceptions can also be thrown by the C++ run-time environment should it encounter an unexpected condition like running out of memory. A thrown exception is "***caught***" by other code that "handles" the exception somehow, or the program is terminated unexpectedly. (We say more about catching exceptions shortly.)

Exceptions are a relatively recent addition to C++. Prior to having exceptions, errors were typically handled by having the program abort at the source of the error or by having the involved function return some special value. Exceptions provide a much cleaner mechanism for handling errors. Nevertheless, for historical reasons, many of the functions in the C++ standard library do not throw exceptions. Typically they return some sort of special error status, or set an error flag, which can be tested.

Exceptions are thrown when a piece of code finds some sort of problem during execution. Since there are many types of possible errors, when an exception is thrown, it is identified by a type. Typically this type is a class whose members provide information as to the exact nature of the error, for example a string containing a descriptive error message.

Exception types often form hierarchies. For example, let's imagine a hypothetical mathematics library, which may generate many different types of errors. The library might begin by defining one generic exception, MathException, representing all types of mathematical errors, and then derive more specific exceptions for particular error conditions. The *errMsg* member holds a message string with an informative message. Here is a possible definition of this generic class.

```
class MathException {                                  // generic math exception
public:
  MathException(const string& err)                     // constructor
    : errMsg(err) { }
  string getError() { return errMsg; }                 // access error message
private:
  string errMsg;                                       // error message
};
```

Using Inheritance to Define New Exception Types

The above MathException class would likely have other member functions, for example, for accessing the error message. We may then add more specific exceptions, such as ZeroDivide, to handle division by zero, and NegativeRoot, to handle attempts to compute the square root of a negative number. We could use class inheritance to represent this hierarchical relationship, as follows.

```
class ZeroDivide : public MathException {
public:
  ZeroDivide(const string& err)                 // divide by zero
  : MathException(err) { }
};

class NegativeRoot : public MathException {
public:
  NegativeRoot(const string& err)               // negative square root
  : MathException(err) { }
};
```

2.4.2 Throwing and Catching Exceptions

Exceptions are typically processed in the context of "try" and "catch" blocks. A ***try block*** is a block of statements proceeded by the keyword **try**. After a try block, there are one or more ***catch blocks***. Each catch block specifies the type of exception that it catches. Execution begins with the statements of the try block. If all goes smoothly, then execution leaves the try block and skips over its associated catch blocks. If an exception is thrown, then the control immediately jumps into the appropriate catch block for this exception.

For example, suppose that we were to use our mathematical library as part of the implementation of a numerical application. We would enclose the computations of the application within a try block. After the try block, we would catch and deal with any exceptions that arose in the computation.

```
try {
  // . . . application computations
  if (divisor == 0)                             // attempt to divide by 0?
    throw ZeroDivide("Divide by zero in Module X");
}
catch (ZeroDivide& zde) {
  // handle division by zero
}
catch (MathException& me) {
  // handle any math exception other than division by zero
}
```

Processing the above try block is done as follows. The computations of the try block are executed. When an attempt is discovered to divide by zero, ZeroDivide is thrown, and execution jumps immediately to the associated **catch** statement where corrective recovery and clean up should be performed.

Let us study the entire process in somewhat greater detail. The **throw** statement is typically written as follows:

```
throw exception_name(arg1, arg2, ...)
```

where the arguments are passed to the exception's constructor.

Exceptions may also be thrown by the C++ run-time system itself. For example, if an attempt to allocate space in the free store using the **new** operator fails due to lack of space, then a bad_alloc exception is thrown by the system.

When an exception is thrown, it must be ***caught*** or the program will abort. In any particular function, an exception in that function can be passed through to the calling function or it can be caught in that function. When an exception is caught, it can be analyzed and dealt with. The general syntax for a ***try-catch block*** in C++ is as follows:

```
try
    try_statements
catch ( exception_type_1 identifier_1 )
    catch_statements_1
...
catch ( exception_type_n identifier_n )
    catch_statements_n
```

Execution begins in the "*try_statements*." If this execution generates no exceptions, then the flow of control continues with the first statement after the last line of the entire try-catch block. If, on the other hand, an exception is generated, execution in the try block terminates at that point and execution jumps to the first catch block matching the exception thrown. Thus, an exception thrown for a derived class will be caught by its base class. For example, if we had thrown NegativeRoot in the example above, it would be caught by catch block for MathException. Note that because the system executes the first matching catch block, exceptions should be listed in order of most specific to least specific. The special form "catch(...)" catches ***all*** exceptions.

The "*identifier*" for the catch statement identifies the exception object itself. As we said before, this object usually contains additional information about the exception, and this information may be accessed from within the catch block. As is common in passing class arguments, the exception is typically passed as a reference or a constant reference. Once execution of the catch block completes, control flow continues with the first statement after the last catch block.

The recovery action taken in a catch block depends very much on the particular application. It may be as simple as printing an error message and terminating the program. It may require complex clean-up operations, such as deallocating dynamically allocated storage and restoring the program's internal state. There are also some interesting cases in which the best way to handle an exception is to ignore it (which can be specified by having an empty catch block). Ignoring an exception is usually done, for example, when the programmer does not care whether there was an exception or not. Another legitimate way of handling exceptions is to throw another exception, possibly one that specifies the exceptional condition more precisely.

2.4.3 Exception Specification

When we declare a function, we should also specify the exceptions it might throw. This convention has both a functional and courteous purpose. For one, it lets users know what to expect. It also lets the compiler know which exceptions to prepare for. The following is an example of such a function definition.

```
void calculator() throw(ZeroDivide, NegativeRoot) {
    // function body ...
}
```

This definition indicates that the function calculator (and any other functions it calls) can throw these two exceptions or exceptions derived from these types, but no others.

By specifying all the exceptions that might be thrown by a function, we prepare others to be able to handle all of the exceptional cases that might arise from using this function. Another benefit of declaring exceptions is that we do not need to catch those exceptions in our function, which is appropriate, for example, in the case where other code is responsible for causing the circumstances leading up to the exception.

The following illustrates an exception that is "passed through."

```
void getReadyForClass() throw(ShoppingListTooSmallException,
                                        OutOfMoneyException) {
  goShopping();  // I don't have to try or catch the exceptions
                 // which goShopping() might throw because
                 // getReadyForClass() will just pass these along.
  makeCookiesForTA();
}
```

A function can declare that it throws as many exceptions as it likes. Such a listing can be simplified somewhat if all exceptions that can be thrown are derived classes of the same exception. In this case, we only have to declare that a function throws the appropriate base class.

Suppose that a function does not contain a **throw** specification. It would be natural to assume that such a function does not throw any exceptions. In fact, it has quite a different meaning. If a function does not provide a **throw** specification, then it may throw ***any*** exception. Although this is confusing, it is necessary to maintain compatibility with older versions of C++. To indicate that a function throws no exceptions, provide the **throw** specifier with an empty list of exceptions.

```
void func1();                       // can throw any exception
void func2() throw();               // can throw no exceptions
```

Generic Exception Class

We declare many different exceptions in this book. In order to structure these exceptions hierarchically, we need to have one generic exception class that serves as the "mother of all exceptions." C++ does not provide such a generic exception, so we created one of our own. This class, called RuntimeException, is shown below. It has an error message as its only member. It provides a constructor that is given an informative error message as its argument. It also provides a member function getMessage that allows us to access this message.

```
class RuntimeException {            // generic run-time exception
private:
  string errorMsg;
public:
  RuntimeException(const string& err) { errorMsg = err; }
  string getMessage() const { return errorMsg; }
};
```

By deriving all of our exceptions from this base class, for any exception *e*, we can output *e*'s error message by invoking the inherited getMessage function.

2.5 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-2.1 What are some potential efficiency disadvantages of having very deep inheritance trees, that is, a large set of classes, A, B, C, and so on, such that B extends A, C extends B, D extends C, etc.?

R-2.2 What are some potential efficiency disadvantages of having very shallow inheritance trees, that is, a large set of classes, A, B, C, and so on, such that all of these classes extend a single class, Z?

R-2.3 Give three examples of life-critical software applications.

R-2.4 Give an example of a software application where adaptability can mean the difference between a prolonged sales lifetime and bankruptcy.

R-2.5 Describe a component from a text-editor GUI (other than an "edit" menu) and the member functions that it encapsulates.

R-2.6 Draw a class inheritance diagram for the following set of classes.

- Class Goat extends Object and adds a member variable *tail* and functions milk and jump.
- Class Pig extends Object and adds a member variable *nose* and functions eat and wallow.
- Class Horse extends Object and adds member variables *height* and *color*, and functions run and jump.
- Class Racer extends Horse and adds a function race.
- Class Equestrian extends Horse and adds a member variable *weight* and functions trot and isTrained.

R-2.7 A derived class's constructor explicitly invokes its base class's constructor, but a derived class's destructor cannot invoke its base class's destructor. Why does this apparent asymmetry make sense?

R-2.8 Give a short fragment of C++ code that uses the progression classes from Section 2.2.3 to find the 7th value of a Fibonacci progression that starts with 3 and 4 as its first two values.

R-2.9 If we choose $inc = 128$, how many calls to the nextValue function from the ArithProgression class of Section 2.2.3 can we make before we cause a long-integer overflow, assuming a 64-bit long integer?

R-2.10 Suppose we have a variable *p* that is declared to be a pointer to an object of type Progression using the classes of Section 2.2.3. Suppose further that *p* actually points to an instance of the class GeomProgression that was created with the default constructor. If we cast *p* to a pointer of type Progression and call *p*->firstValue(), what will be returned? Why?

R-2.11 Consider the inheritance of classes from Exercise R-2.6, and let *d* be an object variable of type Horse. If *d* refers to an actual object of type Equestrian, can it be cast to the class Racer? Why or why not?

R-2.12 Generalize the Person-Student class hierarchy to include classes Faculty, UndergraduateStudent, GraduateStudent, Professor, Instructor. Explain the inheritance structure of these classes, and derive some appropriate member variables for each class.

R-2.13 Give an example of a C++ code fragment that performs an array reference that is possibly out of bounds, and if it is out of bounds, the program catches that exception and prints an appropriate error message.

R-2.14 Consider the following code fragment:

```
class Object
  { public: virtual void printMe() = 0; };
class Place : public Object
  { public: virtual void printMe() { cout << "Buy it.\n"; } };
class Region : public Place
  { public: virtual void printMe() { cout << "Box it.\n"; } };
class State : public Region
  { public: virtual void printMe() { cout << "Ship it.\n"; } };
class Maryland : public State
  { public: virtual void printMe() { cout << "Read it.\n"; } };

int main() {
  Region*   mid = new State;
  State*    md  = new Maryland;
  Object*   obj = new Place;
  Place*    usa = new Region;
  md->printMe();
  mid->printMe();
  (dynamic_cast<Place*>(obj))->printMe();
  obj = md;
  (dynamic_cast<Maryland*>(obj))->printMe();
  obj = usa;
  (dynamic_cast<Place*>(obj))->printMe();
  usa = md;
  (dynamic_cast<Place*>(usa))->printMe();
  return EXIT_SUCCESS;
}
```

What is the output from calling the main function of the Maryland class?

R-2.15 Write a short C++ function that counts the number of vowels in a given character string.

R-2.16 Write a short C++ function that removes all the punctuation from a string s storing a sentence. For example, this operation would transform the string `"Let's try, Mike."` to `"Lets try Mike"`.

R-2.17 Write a short program that takes as input three integers, a, b, and c, and determines if they can be used in a correct arithmetic formula (in the given order), like "$a+b=c$," "$a=b-c$," or "$a*b=c$."

R-2.18 Write a short C++ program that creates a Pair class that can store two objects declared as generic types. Demonstrate this program by creating and printing Pair objects that contain five different kinds of pairs, such as <int,string> and <float,long>.

Creativity

C-2.1 Give an example of a C++ program that outputs its source code when it is run. Such a program is called a ***quine***.

C-2.2 Suppose you are on the design team for a new e-book reader. What are the primary classes and functions that the C++ software for your reader will need? You should include an inheritance diagram for this code, but you don't need to write any actual code. Your software architecture should at least include ways for customers to buy new books, view their list of purchased book, and read their purchased books.

C-2.3 Most modern C++ compilers have optimizers that can detect simple cases when it is logically impossible for certain statements in a program to ever be executed. In such cases, the compiler warns the programmer about the useless code. Write a short C++ function that contains code for which it is provably impossible for that code to ever be executed, but your favorite C++ compiler does not detect this fact.

C-2.4 Design a class Line that implements a line, which is represented by the formula $y=ax+b$. Your class should store a and b as **double** member variables. Write a member function intersect(ℓ) that returns the x coordinate at which this line intersects line ℓ. If the two lines are parallel, then your function should throw an exception Parallel. Write a C++ program that creates a number of Line objects and tests each pair for intersection. Your program should print an appropriate error message for parallel lines.

C-2.5 Write a C++ class that is derived from the Progression class to produce a progression where each value is the absolute value of the difference between the previous two values. You should include a default constructor that starts with 2 and 200 as the first two values and a parametric constructor that starts with a specified pair of numbers as the first two values.

C-2.6 Write a C++ class that is derived from the Progression class to produce a progression where each value is the square root of the previous value. (Note that you can no longer represent each value with an integer.) You should include a default constructor that starts with 65,536 as the first value and a parametric constructor that starts with a specified (**double**) number as the first value.

C-2.7 Write a program that consists of three classes, A, B, and C, such that B is a subclass of A and C is a subclass of B. Each class should define a member variable named "x" (that is, each has its own variable named x). Describe a way for a member function in C to access and set A's version of x to a given value, without changing B or C's version.

C-2.8 Write a set of C++ classes that can simulate an Internet application, where one party, Alice, is periodically creating a set of packets that she wants to send to Bob. The Internet process is continually checking if Alice has any packets to send, and if so, it delivers them to Bob's computer, and Bob is periodically checking if his computer has a packet from Alice, and, if so, he reads and deletes it.

C-2.9 Write a C++ program that can input any polynomial in standard algebraic notation and outputs the first derivative of that polynomial.

Projects

P-2.1 Write a C++ program that can take a positive integer greater than 2 as input and write out the number of times one must repeatedly divide this number by 2 before getting a value less than 2.

P-2.2 Write a C++ program that "makes change." Your program should input two numbers, one that is a monetary amount charged and the other that is a monetary amount given. It should return the number of each kind of bill and coin to give back as change for the difference between the amounts given and charged. The values assigned to the bills and coins can be based on the monetary system of any government. Try to design your program so that it returns the fewest number of bills and coins as possible.

P-2.3 Implement a templated C++ class Vector that manipulates a numeric vector. Your class should be templated with any numerical scalar type T, which supports the operations $+$ (addition), $-$ (subtraction), and $*$ (multiplication). In addition, type T should have constructors $T(0)$, which produces the additive identity element (typically 0) and $T(1)$, which produces the multiplicative identity (typically 1). Your class should provide a constructor, which is given the size of the vector as an argument. It should provide member functions (or operators) for vector addition, vector subtraction, multiplication of a scalar and a vector, and vector dot product.

Write a class Complex that implements a complex number by overloading the operators for addition, subtraction, and multiplication. Implement three concrete instances of your class Vector with the scalar types **int**, **double**, and Complex, respectively.

P-2.4 Write a simulator as in the previous project, but add a Boolean gender field and a floating-point strength field to each Animal object. Now, if two animals of the same type try to collide, then they only create a new instance of that type of animal if they are of different genders. Otherwise, if two animals of the same type and gender try to collide, then only the one of larger strength survives.

P-2.5 Write a C++ program that has a Polygon interface that has abstract functions, area(), and perimeter(). Implement classes for Triangle, Quadrilateral, Pentagon, Hexagon, and Octagon, which implement this interface, with the obvious meanings for the area() and perimeter() functions. Also implement classes, IsoscelesTriangle, EquilateralTriangle, Rectangle, and Square, which have the appropriate inheritance relationships. Finally, write a simple user interface that allows users to create polygons of the various types, input their geometric dimensions, and then output their area and perimeter. For extra effort, allow users to input polygons by specifying their vertex coordinates and be able to test if two such polygons are similar.

P-2.6 Write a C++ program that inputs a document and then outputs a bar-chart plot of the frequencies of each alphabet character that appears in that document.

P-2.7 Write a C++ program that inputs a list of words separated by whitespace, and outputs how many times each word appears in the list. You need not worry about efficiency at this point, however, as this topic is something that will be addressed later in this book.

Chapter Notes

For a broad overview of developments in computer science and engineering, we refer the reader to *The Computer Science and Engineering Handbook* [96]. For more information about the Therac-25 incident, please see the paper by Leveson and Turner [63].

The reader interested in studying object-oriented programming further, is referred to the books by Booch [13], Budd [16], and Liskov and Guttag [68]. Liskov and Guttag [68] also provide a nice discussion of abstract data types, as does the survey paper by Cardelli and Wegner [19] and the book chapter by Demurjian [27] in the *The Computer Science and Engineering Handbook* [96]. Design patterns are described in the book by Gamma *et al.* [35]. The class inheritance diagram notation we use is derived from the Gamma *et al.*