

640

Inheritance for Object-Oriented Design 13

Instead of teaching people that O-O is a type of design, and giving them design principles, people have taught that O-O is the use of a particular tool. We can write good or bad programs with any tool. Unless we teach people how to design, the languages matter very little. The result is that people do bad designs with these languages and get very little value from them.

David Parnas

personal note to Fred Brooks, in *The Mythical Man Month*, Anniversary Edition

In this chapter we'll explore **inheritance**, one of the cornerstones of object-oriented programming. Many experts in programming languages differentiate between *object-based* programming, in which inheritance is not used, and *object-oriented* programming, in which inheritance is used. As you'll see, inheritance makes it possible to reuse classes in a completely different way from what we've seen to this point. The key aspect of inheritance that we'll explore is essentially changing class behavior without having access to the class implementation. This kind of reuse allows companies to design class tool kits for different applications, (e.g., for graphics, networking, or games), and for clients to specialize these classes for their own purposes. The companies designing the tool kits do *not* need to release their implementations, which can be an attractive feature for those who do not want to release proprietary designs or code.

13.1 Essential Aspects of Inheritance

In Chapter 7 we designed and implemented Program 7.8, *quiz.cpp*, for giving students different kinds of quizzes. We developed classes for two different kinds of quiz questions: an arithmetic quiz question about simple addition problems (see Program 7.6, *mathquest.h*) and a geography quiz question about U.S. states and their capitals (see *capquest.h*.) By using the same class name, `Question`, for both different kinds of quiz question, we made it possible to reuse the same quiz program as well as the classes `Quiz` and `Student` defined in the quiz program.

However, to have different quizzes we had to change the preprocessor include directive from `#include "mathquest.h"` to `#include "capquest.h"` and recompile the program. In this chapter we'll study **inheritance**, a programming technique that permits a common interface to be *inherited* or reused by many classes. Client programs written to conform to the interface can be used with any of the interface-inheriting classes. Programs do not, necessarily, need to be recompiled to use more than one of the conforming classes. As we'll see, several different quiz questions can be used in the same program when we use inheritance.

13.1.1 The Inheritance Hierarchy for Streams

You've already used inheritance in many of the C++ programs you've written, although you probably haven't been explicitly aware of doing so. The stream hierarchy of classes uses inheritance so that you can write a function with an `istream` parameter, but pass as arguments, `cin`, an `ifstream` variable, or an `istringstream` variable. This use of streams is shown in *streaminherit.cpp*, Program 13.1

Program 13.1 *streaminherit.cpp*

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
using namespace std;
#include "prompt.h"

// show stream inheritance

void doInput(istream& input)
// precondition: there are three strings to read in input
{
    string s;
    int k;
    for(k=0; k < 3; k++)
    {   input >> s;
        cout << k << ".\t" << s << endl;
    }
    cout << endl;
}

int main()
{
    string filename = PromptString("filename: ");
    ifstream input(filename.c_str());
    string firstline;
    getline(input,firstline);           // first line of input file
    istringstream linestream(firstline); // stream bound to first line

    cout << "first three words on first line are\n--" << endl;
    doInput(linestream);

    cout << "first three words on second line are\n--" << endl;
    doInput(input);

    cout << "first three words from keyboard are\n--" << endl;
    doInput(cin);
    return 0;
}
```

streaminherit.cpp

```

                                O U T P U T
prompt> poe.txt
first three words on first line are
-----
0.      The
1.      Cask
2.      of

first three words on second line are
---
0.      Edgar
1.      Allan
2.      Poe

first three words from keyboard are
---
this is a test of reading from the keyboard
0.      this
1.      is
2.      a

```

The code in the function `doInput` from *streaminherit.cpp* uses only stream behavior that is common to all input streams, (i.e., extraction using operator `>>`). Other common stream behavior includes input using `get` or `getline` and functions `clear`, `fail`, and `ignore`. By conforming to the common input stream interface, the code is more general since it can be used with any input stream. This includes input stream classes that aren't yet written, but that when written will conform to the common stream interface by using the inheritance mechanism discussed in this chapter. If the function `doInput` used the stream function `seekg` (see How to B) to reset the stream to the beginning, then unexpected behavior will result when `cin` is passed since the standard input stream `cin` is not a *seekable input stream* as are `ifstream` and `istringstream` streams.¹ If `doInput` uses `seekg` the code is not conforming to the common interface associated with all streams (the `seekg` function can be applied to `cin`, but the application doesn't do anything).



¹A seekable input stream can be reread by moving or *seeking* the location of input to the beginning (or end, or sometimes middle). The standard input stream isn't seekable in the same way a file bound to a text file is seekable.

13.1.2 An Inheritance Hierarchy: Math Quiz Questions

We'll return to the example of giving a computer-based quiz to students. We discussed the development of a program to give quizzes to two students sharing a keyboard in Section 6.2. In this chapter we'll use a simpler quiz program to show the power of inheritance. Because the syntactic details of using inheritance in C++ are somewhat cumbersome, rather than discussing the syntax in detail at first, we'll look at the program, make a modification to it to show what inheritance can do, and then look more at the implementation details.

Program 13.2, *inheritquiz.cpp*, gives an arithmetic quiz to a student. Only one chance is given to get each question correct, and no score is kept. Three different kinds of questions are used in the program, ranging in difficulty from easy to hard (depending, of course, on your point of view.) The program is a **prototype** to show what can be done, but is not a finished product (this program tip repeats Tip 6.6).

Program Tip 13.1: A prototype is a good way to start the implementation phase of program development and to help in the design process. A prototype is a “realistic model of a system’s key functions” [McC93]. Booch says that “prototypes are by their very nature incomplete and only marginally engineered.” [Boo94] A prototype is an aid to help find some of the important issues before design and implementation are viewed as frozen, or unchanging. For those developing commercial software, prototypes can help clients articulate their needs better than a description in English.

As we'll see when we explore implementation details of using inheritance in C++, pointers to objects are often used rather than objects themselves. All our uses of inheritance will use pointers or references; both are used in *inheritquiz.cpp*.²

Program 13.2 *inheritquiz.cpp*

```
#include <iostream>
#include <string>
using namespace std;

#include "tvector.h"
#include "prompt.h"
#include "randgen.h"
#include "mathquestface.h"

// prototype quiz program for demonstrating inheritance

void GiveQuestion(MathQuestion& quest)
// post: quest is asked once, correct response is given
{
    string answer;
```

²All our examples of inheritance use **polymorphism**, which we'll define later. Polymorphism requires either a pointer or a reference.

```

    cout << endl << quest.Description() << endl;
    cout << "type answer after the question" << endl;
    quest.Create();
    quest.Ask();
    cin >> answer;
    if (quest.IsCorrect(answer))
    {
        cout << "Excellent!, well done" << endl;
    }
    else
    {
        cout << "I'm sorry, the answer is " << quest.Answer() << endl;
    }
}

int main()
{
    tvector<MathQuestion *> questions;          // fill with questions

    questions.push_back(new MathQuestion());
    questions.push_back(new CarryMathQuestion());
    questions.push_back(new HardMathQuestion());

    int qCount = PromptRange("how many questions",1,10);
    RandGen gen;
    int k;
    for(k=0; k < qCount; k++)
    {
        int index = gen.RandInt(0,questions.size()-1);
        GiveQuestion(*questions[index]);
    }
    for(k=0; k < questions.size(); k++)
    {
        delete questions[k];
    }
    return 0;
}

```

inheritquiz.cpp

The parameter to the function `GiveQuestion` is a `MathQuestion` passed by reference. Based on the declaration and initialization of the vector `questions`, it appears that each vector element holds a pointer to a `MathQuestion`, but that the pointers actually point to a `MathQuestion`, a `CarryMathQuestion`, and a `HardMathQuestion`, respectively for indexes 0, 1, and 2.

Examining the sample output that follows shows that three different kinds of question are, in fact, asked during one run of the program. Looking at the program carefully will help us develop some questions that will guide the discussion of inheritance, how it works, and how it is implemented in C++.

1. How can a pointer to `MathQuestion` actually point to some other type of object (the other kinds of questions).
2. How can different objects (dereferenced by the `*` in the `GiveQuestion` call) be passed to `GiveQuestion` which expects a `MathQuestion` by reference?
3. How are different questions actually created by the call `quest.Create()` in `GiveQuestion`.
4. How can we develop another kind of question and add it to the program?

```

                                O U T P U T

prompt> inheritquiz
how many questions between 1 and 10: 4

addition of three-digit numbers
type answer after the question
  134
+ 122
-----
  256
Excellent, well done!

addition of three-digit numbers
type answer after the question
  175
+ 192
-----
  267
I'm sorry, the answer is 367

addition of two-digit numbers with NO carry
type answer after the question
   11
+  18
-----
   29
Excellent, well done!

addition of two-digit numbers with a carry
type answer after the question
   38
+  39
-----
   77
Excellent, well done!

```

To answer the four questions raised above, we'll look at inheritance conceptually, and how it is implemented in C++. The example of stream inheritance in Program 13.1, *streaminherit.cpp*, showed that a common interface allows objects with different types to be used in the same way, by the same code. The math quiz program, *inheritquiz.cpp* leverages a common interface in the same way. Each of the three question types stored

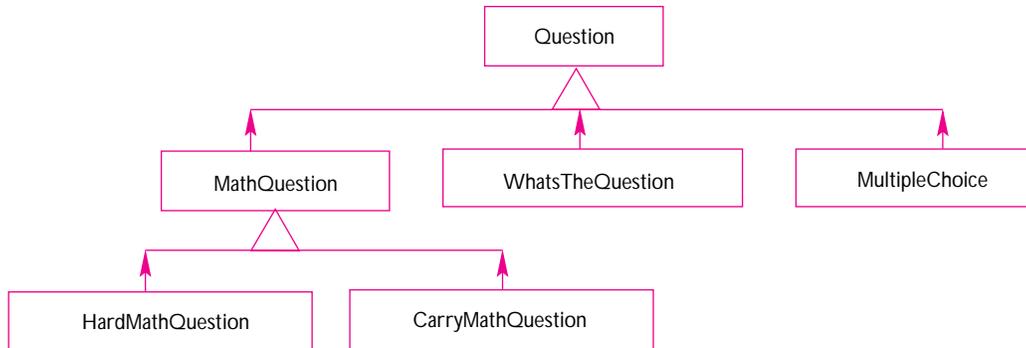


Figure 13.1 Hierarchy of math and other quiz questions.

in the vector `question` **is a kind of** `MathQuestion`. This **is-a** relationship is conceptual and realized in code. Conceptually, clients write quiz programs with code that conforms to the `MathQuestion` class interface and expect to use `MathQuestion` objects. For example, the function `GiveQuestion` has a `MathQuestion` parameter. Objects that are instances of other classes that inherit from `MathQuestion`, like `CarryMathQuestion`, can be used as though they were `MathQuestion` objects, (e.g., they can be elements of a vector like `question` in `main`).

An inheritance hierarchy, like the one illustrated in Figure 13.1, models an is-a relationship, where is-a means “has the same behavior” and “can be used in place of-a or as-a.” This kind of hierarchy is realized in C++ by declaring **subclasses**, also called **derived classes**, to inherit from a **base class**, also called a **super class**. In Figure 13.1 the classes `CarryMathQuestion` and `HardMathQuestion` derive from the super class `MathQuestion`. However, `MathQuestion` itself is a derived or subclass of the super/base class `Question`. Derived classes have functions with the same names as functions in the super class, (e.g., the functions `Create`, `Ask`, and `Answer` for the `Question` hierarchy). These functions can have different behavior in each subclass which is what makes the different kinds of quiz question in the same program possible.

13.1.3 Implementing Inheritance

The interfaces for the three different kinds of `MathQuestion` classes are declared in `mathquestface.h`, Program 13.3. We’ll use four new syntactic constructs in C++ to implement an inheritance hierarchy. Three of the new syntactic constructs are shown in `mathquestface.h`, the fourth, the abstract base class, is discussed in Section 13.2.

- public inheritance
- virtual functions
- protected data members (and functions)

 Program 13.3 mathquestface.h

```

#ifndef _MATHQUESTFACE_H
#define _MATHQUESTFACE_H

// quiz questions involving arithmetic (addition)
// see comments in questface.h for the naming conventions
// used in quiz classes
//
// MathQuestion()      - no carry involved, two-digit numbers
// CarryMathQuestion() - does have carry, two-digit numbers
// HardMathQuestion()  - three digit addition
//
// these classes add method Description() to the question hierarchy

#include "questface.h"

class MathQuestion : public Question
{
public:
    MathQuestion();
    virtual bool    IsCorrect(const string& answer) const;
    virtual string  Answer()                const;
    virtual void    Ask()                   const;
    virtual string  Description()           const;

    virtual void    Create(); // create a new question

protected:
    string myAnswer; // store the answer as a string here
    int myNum1;      // numbers used in question
    int myNum2;
};

class CarryMathQuestion : public MathQuestion
{
public:
    CarryMathQuestion();
    virtual string Description() const;
    virtual void    Create();
};

class HardMathQuestion : public MathQuestion
{
public:
    HardMathQuestion();
    virtual string Description() const;
    virtual void    Create();
};

#endif

```

mathquestface.h

13.1.4 Public Inheritance

In *mathquestface.h*, the subclasses `CarryMathQuestion` and `HardMathQuestion` each express their dependence on the superclass from which they're derived. This dependency is shown on the first line of each class declaration.

```
class CarryMathQuestion : public MathQuestion
{...
};
class HardMathQuestion : public MathQuestion
{...
};
```

In general, each subclass expresses a dependency relationship to a superclass by using the keyword **public** and the name of the superclass. This is called **public inheritance** (we will not use private inheritance, virtual inheritance, or any of the other kinds of inheritance that are possible in C++.) As we'll see, a subclass inherits an

Syntax: public inheritance

```
class Subclass : public Superclass
{
    methods and instance variables
};
```

interface from its superclass, and can inherit behavior (member functions) too. Inheritance is a chained, or transitive relationship so that if subclass C inherits from superclass B, but B is itself a subclass of superclass A, then C inherits from A as well, al-

though this is not shown explicitly in the declaration of C. In Figure 13.1, the class `HardMathQuestion` is a subclass of both `MathQuestion` and `Question`.

As we mentioned earlier, inheritance models an is-a relationship. When a subclass B inherits from a superclass A, any object of type B can be used where an object of type A is expected. This makes it possible, for example, to pass a `HardMathQuestion` object to the function `GiveQuestion` which has a parameter of type `MathQuestion`. However, as we use inheritance, the is-a relationship captured by public inheritance will *not* work correctly in a C++ program unless references or pointers are used. An object, say `hmq`, that's an instance of the `HardMathQuestion` class can be passed as a parameter to a function expecting a `MathQuestion` object *only* if `hmq` is passed by reference or as a pointer. Similarly `hmq` can be assigned as a `MathQuestion` object *only* if the assignment of `hmq` uses a pointer to `hmq`. In *inheritquiz.cpp* objects are passed by reference to `GiveQuestion` and the vector `question` holds pointers to `MathQuestion` objects so that inheritance will work as intended. We'll study why this restriction is necessary in Section 13.1.5.

Program Tip 13.2: Public inheritance should model an is-a relationship.

For is-a to work as expected, objects in an inheritance hierarchy should be passed by reference or as pointers and assigned using pointers whenever a subclass object is used as a superclass object.

13.1.5 Virtual Functions

Inheritance is exploited in *inheritquiz.cpp* since we can pass any kind of math quiz-question to the function `GiveQuestion` and different questions are created depending on the type of the object passed. From the compiler's perspective, the parameter to `GiveQuestion` is a reference to a `MathQuestion` object. How does the compiler know to call `HardMathQuestion::Create` when a hard question object is passed and to call `CarryMathQuestion::Create` when a carry question object is passed?

The compiler does not determine which function to call at **compile time** (when the program is compiled), but delays choosing which function to call until **run time** (when the program is executing or running). At run time different objects can be passed to `GiveQuestion`. Although the compiler thinks of the parameter as having type `MathQuestion`, the actual type may be different because of inheritance. The compiler calls the "right" version of `Create` because the function `Create` is a **virtual function**. Virtual functions are called **polymorphic** functions because they can take many forms³ depending on the run time type of an object rather than the compile time type of an object.

What does all that really mean? It means that if you put the key word *virtual* before a member function in a superclass, then the member function that's called will be the subclass version of the member function if a subclass is what's actually used when the program is running. In *inheritquiz.cpp*, Program 13.2, different kinds of question are created because the member functions `Create` in each of the three classes in the math question hierarchy are different, as you can see in *mathquestface.cpp*, Program 13.4. We'll discuss when to make functions virtual in Section 13.2.2.

Program Tip 13.3: The keyword *virtual* is not required in subclasses, but it's good practice to include it as needed in each subclass. Any member function that is virtual in a superclass is also virtual in a derived class. Since a subclass may be a superclass at some point (e.g., as `MathQuestion` is a subclass of `Question` but a superclass of `HardMathQuestion`), including the word *virtual* every time a member function is declared is part of safe programming.

As you can see in the definitions of each member function in *mathquestface.cpp*, the word *virtual* appears only in the interface, or `.h` file, not in the implementation or `.cpp` file. Note that the constructors for `CarryMathQuestion` and `HardMathQuestion` each explicitly call the superclass constructor `MathQuestion()`. A superclass constructor will always be called from a subclass, even if the compiler must generate an implicit call. As we'll see in Section 13.2 some classes cannot be constructed which is why `MathQuestion` does not call the constructor for `Question`, its superclass.

³The word polymorphic is derived from the Greek words *polus*, (many) and *morphe*, (shape).

Program Tip 13.4: Each subclass should explicitly call the constructor of its superclass. The constructor will be called automatically if you don't include an explicit call, and sometimes parameters should be included in the superclass constructor. Superclass constructors must be called from an initializer list, not from the body of the subclass constructor. If the superclass is an *abstract base class* (see Section 13.2) no superclass constructor can be called.

Program 13.4 `mathquestface.cpp`

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "mathquestface.h"
#include "randgen.h"
#include "strutils.h"

MathQuestion::MathQuestion()
    : myAnswer("*** error ***"),
      myNum1(0),
      myNum2(0)
{
    // nothing to initialize
}

void MathQuestion::Create()
{
    RandGen gen;
    // generate random numbers until there is no carry
    do
    {
        myNum1 = gen.RandInt(10,49);
        myNum2 = gen.RandInt(10,49);
    } while ( (myNum1 % 10) + (myNum2 % 10) >= 10);

    myAnswer = toString(myNum1 + myNum2);
}

void MathQuestion::Ask() const
{
    const int WIDTH = 7;
    cout << setw(WIDTH) << myNum1 << endl;
    cout << "+" << setw(WIDTH-1) << myNum2 << endl;
    cout << "---" << endl;
    cout << setw(WIDTH-myAnswer.length()) << " ";
}

bool MathQuestion::IsCorrect(const string& answer) const
{
    return myAnswer == answer;
}
```

```
string MathQuestion::Answer() const
{
    return myAnswer;
}

string MathQuestion::Description() const
{
    return "addition of two-digit numbers with NO carry";
}

CarryMathQuestion::CarryMathQuestion()
    : MathQuestion()
{
    // all done in base class constructor
}

void CarryMathQuestion::Create()
{
    RandGen gen;
    // generate random numbers until there IS a carry
    do
    {
        myNum1 = gen.RandInt(10,49);
        myNum2 = gen.RandInt(10,49);
    } while ( (myNum1 % 10) + (myNum2 % 10) < 10);

    myAnswer = toString(myNum1 + myNum2);
}

string CarryMathQuestion::Description() const
{
    return "addition of two-digit numbers with a carry";
}

HardMathQuestion::HardMathQuestion()
    : MathQuestion()
{
    // all done in base class constructor
}

void HardMathQuestion::Create()
{
    RandGen gen;
    myNum1 = gen.RandInt(100,200);
    myNum2 = gen.RandInt(100,200);
    myAnswer = toString(myNum1 + myNum2);
}

string HardMathQuestion::Description() const
{
    return "addition of three-digit numbers";
}
```

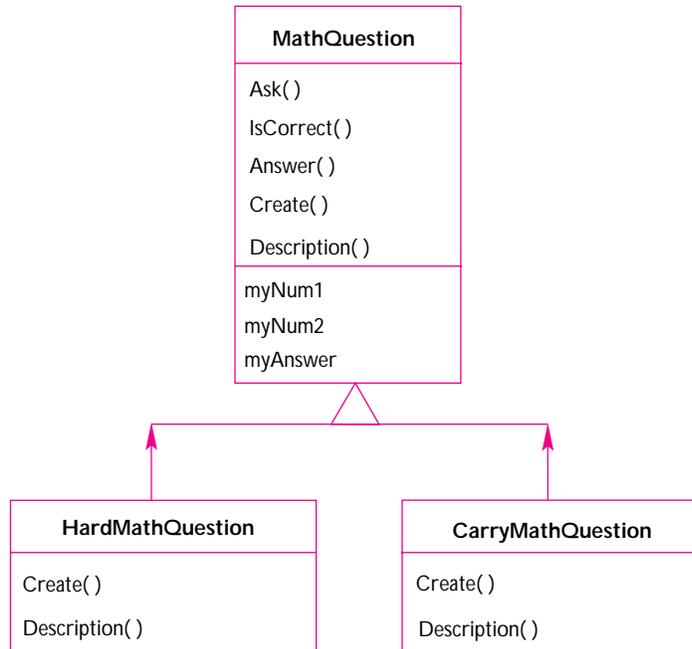


Figure 13.2 Overriding functions in the `MathQuestion` hierarchy.

Overriding inherited functions. In `mathquestface.h`, Program 13.3 the declarations for `CarryMathQuestion` and `HardMathQuestion` include only a constructor and prototypes for `Create` and `Description`. Declarations for the other inherited, virtual functions `IsCorrect`, `Ask`, and `Answer` are *not included*. This relationship is diagrammed in Figure 13.2.

Nevertheless, it's possible to call `CarryMathQuestion::Ask()` and have an addition question that requires carrying asked of the user. The functions `Create` and `Description` are included in the interfaces of the derived classes and implemented in `mathquestface.cpp` because their behavior is different from what's inherited from the superclass versions of these functions. When a subclass implements an inherited function it is called **overriding** the inherited function. When a method is overridden in a subclass, the subclass uses its own version of the method rather than the inherited version. Sometimes an inherited function works well even in subclasses. This is the case with the inherited functions `Ask`, `Answer`, and `IsCorrect`. The behavior, or implementation, of each of these functions does not need to be changed or **specialized** in the subclasses, so these inherited functions are not overridden. Since they're inherited unchanged, the functions do not appear in the declarations of the derived classes.

13.1.6 Protected Data Members

A subclass inherits more than behavior from its superclass, but also inherits state. This means that in addition to inheriting member function interfaces and, sometimes, implementations; a subclass inherits instance variables. Private instance variables are accessible only within member functions of the class in which the variables are declared; private data of a superclass are *not* accessible in any derived class. The private data are present in the derived classes, and if there are accessor functions or mutator functions inherited from the superclass these inherited functions can be used to access the private data, but no derived class member functions can access the private data directly.

In some inheritance hierarchies it makes sense for derived classes to access the instance variables that make up the state. In the math question hierarchy, for example, the functions `Create` assign values to `myNum1` and `myNum2` and these values are used in the functions `Ask` (although `Ask` is not overridden by the derived classes.) Any variables and functions that are declared as **protected** are accessible to the member functions of the class in which they're declared as protected, but also to the member functions of all derived classes. The instance variables `myNum1`, `myNum2`, and `myAnswer` are all declared as protected, so they are accessible in `MathQuestion` methods and also in the derived classes `HardMathQuestion` and `CarryMathQuestion`. This is diagrammed in Figure 13.2.

It's often a good idea to avoid inheriting state, and to inherit only interface and behavior. The problems that arise from inheriting state invariably stem from trying to inherit from more than one class, so-called **multiple inheritance**. We won't use multiple inheritance in this chapter, although we do use it in conjunction with the graphics package discussed in How to H.



ProgramTip 13.5: When possible, inherit only interface and behavior, not state. Minimize the inheritance of state if you think you'll eventually need to inherit behavior or interfaces from more than one class. When you're designing an inheritance hierarchy, protected data are accessible in derived classes, but private data are not, although the private data are present.

Pause to Reflect



13.1 If the call to `new` is not included in each call to `push_back` in Program 13.2, `inheritquiz.cpp`, will the program compile? Why?

13.2 Suppose the dereferencing operator isn't used in the call to `GiveQuestion` from `main` in `inheritquiz.cpp`. Explain how `GiveQuestion` should be modified so that it works with this call: `GiveQuestion(questions[index])`.

13.3 Suppose you create a new class named `MultMathProblem` for quiz questions based on multiplying a one-digit number by a two-digit number. Explain why the method `Ask` should be overridden in this class although it wasn't necessary to override it in the addition quiz questions.

13.4 Arguably, the behavior of the `Description` function in each of the classes in the math question hierarchy is exactly the same, but the string returned differs. The behavior is the same because each function returns a string, but doesn't do anything else different. Explain modifications to the three classes that make up the math question hierarchy so that the description is an argument when the class is constructed and the method `Description` is not overridden in each subclass. The constructor calls in `main` might look like this (arguments are abbreviated.)

```
questions.push_back(
    new MathQuestion("+, 2-digits, no carry"));
questions.push_back(
    new CarryMathQuestion("+, 2-digits, carry"));
questions.push_back(
    new HardMathQuestion("+, 3-digits"));
```

Why is this approach (arguably) not as good as the approach taken in the code (where `Description` is overridden)? Think about what client code should be responsible for and what classes used in client code should be responsible for.

13.5 If `protected` in `MathQuestion` is changed to `private`, the `Create` functions in each subclass will not compile. Why?

13.6 Design a new class for addition of three two-digit numbers with no carry. What inherited methods must you override? Why will you need to add a new data member in the new class? Why is it better to include the data member, say `myNum3` in the new class rather than in the class `MathQuestion`?

13.7 Suppose you add state and behavior to the math question hierarchy so that each question tracks how many times its `Create` method is called. This number should be tracked and updated by the question classes, but readable by client code. In what class(es) should the data and methods go?

13.2 Using an Abstract Base Class

The question hierarchy shown in Figure 13.1 shows other kinds of questions than math questions. The class `WhatsTheQuestion` is designed to encapsulate and generalize the state-capital question generator declared in `capquest.h`, discussed in Section 7.2.6, and implemented in `capquest.cpp`, Program 7.9. The class is generalized because it permits questions like “What’s the capital of Texas?” and “What artist recorded *Slowhand*?” (the answers, respectively, are Austin and Eric Clapton). We’d like to incorporate all these kinds of questions in the same quiz program: “what-the” questions, math questions, and other kinds of questions we haven’t yet designed or implemented.

The new class `WhatsTheQuestion` should not derive from `MathQuestion` if the new class doesn't have anything to do with mathematics. Since inheritance models *is-a* relationships, it would be a mistake to derive a question about state capitals from `MathQuestion` since the state-capital question cannot be used as a math question, and *is-a* means "can be used in a program as a" in our use of inheritance. Instead, we'll create a new abstraction, one that captures the idea of any kind of question. We'll call this new class `Question`; both `MathQuestion` and `WhatsTheQuestion` will derive from `Question` as shown in Figure 13.1.

The class `Question` is an **interface class**. The class exists as a superclass, but primarily as an interface for client programs. Clients write code to the specifications described in prose and code in *questface.h*, Program 13.5. As we'll see, it is not possible to create `Question` objects. Instead, we can create object that are instances of classes that derive from `Question`, and that inherit its interface. Since client programs are written to the interface, the new classes can be used in any client code that uses `Question` objects by reference or as pointers. For example, if we use `Question` instead of `MathQuestion` in the prototype of `GiveQuestion` in Program 13.2, *inheritquiz.cpp*, we'll be able to use the quiz prototype program for all kinds of questions. A new version of the function `GiveQuestion` is shown below. The call to `quest.Description` is commented out and the extraction operator `>>` is replaced by `getline` to allow the user to enter several words in response to a question.

```
void GiveQuestion(Question& quest)
// post: quest is asked once, correct response is given
{
    string answer;
    // cout << endl << quest.Description() << endl;
    cout << "type answer after the question" << endl;
    quest.Create();
    quest.Ask();
    getline(cin, answer);
    if (quest.IsCorrect(answer))
    { cout << "Excellent!, well done" << endl;
    }
    else
    { cout << "I'm sorry, the answer is "
      << quest.Answer() << endl;
    }
}
```

The call of `Description` is commented out because it's a method from the `MathQuestion` hierarchy, but not in our current version of the `Question` hierarchy declared in *questface.h*, Program 13.5.

The declaration for `Question` is like other class declarations, but all the member functions, except the destructor, are declared with `= 0` after the function prototype. As we'll see, these make `Question` an **abstract base class**.

 Program 13.5 questface.h

```

#ifndef _QUESTIONINTERFACE_H
#define _QUESTIONINTERFACE_H

// abstract base class for quiz questions
// derived classes MUST implement four functions:
//
// void Ask()          to ask the question
// string Answer()     to return the answer
// bool IsCorrect(s)  to tell if an answer s is correct
// void Create()       to create a new question
//
// This class conforms to the naming conventions
// of quiz questions in "A Computer Science Tapestry" 2e

#include <string>
using namespace std;

class Question
{
public:
    virtual ~Question() { } // must implement destructor, here inline

    // accessor functions

    virtual bool    IsCorrect(const string& answer) const = 0;
    virtual string  Answer()                          const = 0;
    virtual void    Ask()                              const = 0;

    // mutator functions

    virtual void Create() = 0;
};

#endif

```

questface.h

13.2.1 Abstract Classes and Pure Virtual Functions

A function with = 0 as part of its prototype in a class declaration *must* be overridden in subclasses; such functions are called **pure virtual functions**. The syntax and naming convention are ugly, it's better to think of these functions as **abstract interfaces**. They're abstract because implementations are not provided,⁴ and they're interfaces because subclasses must implement a member function with the same prototype, thus conforming to the interface declared in the base class.

⁴The "must be overridden" rule is correct, but it's possible to supply an implementation of a pure virtual function that can be called from the overriding function in the subclass. However, any class that contains a pure virtual function cannot be instantiated/constructed. For our purposes, pure virtual functions will not have implementations, they're interfaces only.

A class that contains one pure virtual function is called an **abstract base class**, sometimes abbreviated as an **abc** (or, redundantly, an abc class⁵). I'll refer to these as abstract classes. It's not possible to define variables of a type that's an abstract class. Instead, subclasses of the abstract class are designed and implemented. Variables that are instances of these **concrete subclasses** can be defined. A concrete class is one for which variables can be constructed. Concrete is, in general, the opposite of abstract.

Why Use Abstract Classes? Designing an inheritance hierarchy can be tricky. One reason it's tricky is that to be robust, a hierarchy must permit new subclasses to be designed and implemented. Often, the original designer of the hierarchy cannot foresee everything clients will do with the hierarchy. Nevertheless, a well-designed hierarchy will be flexible in both use and modification through subclassing.

One design heuristic that helps make a class hierarchy flexible is to derive only from abstract classes. Clients are forced to implement each pure virtual function and are thus less likely to forget to implement one, thus getting inherited, but unexpected behavior. The hierarchies we show in this book won't cause trouble *as we're using them*. But what about how other programmers will use our hierarchies? In general, you cannot expect all programmers to use your code wisely and not make mistakes. I certainly don't. A lengthy description of why it's a good idea to use abstract classes as superclasses is found in [Mey96] as item 33. This is one of several items that appear in a section called *Programming in the Future Tense*.

Program Tip 13.6: Good software is flexible, robust, and reliable. It meets current needs, but adapts well to future needs, ideally to ideas not completely anticipated when the software is designed and implemented.

Good programmers anticipate that things will change and design code to be adaptable in the face of inevitable change and maintenance.

Program 13.6 `whatsthequizmain.cpp`

```
int main()
{
    tvector<Question *> questions;
    questions.push_back(new HardMathQuestion());
    questions.push_back(new WhatsTheQuestion("what's the capital of ", "statequiz.dat"));
    questions.push_back(new WhatsTheQuestion("what artist made ", "cdquiz.dat"));

    int qCount = PromptLnRange("how many questions", 1, 10);
    RandGen gen;
    for(int k=0; k < qCount; k++)
    { int index = gen.RandInt(0, questions.size()-1);
      GiveQuestion(*questions[index]);
    }
}
```

⁵What does PIN stand for — the thing you type as a password when you use an ATM? There is no such thing as a PIN number, nor an ATM machine. Well, there are such things, but there shouldn't be.

```

}
for(int k=0; k < questions.size(); k++)
{ delete questions[k];
}
return 0;
}

```

whatsthequizmain.cpp

We'll pass different objects to the modified function `GiveQuestion` that uses the `Question` interface. The main that's shown is part of *whatsthequiz.cpp*.⁶

O U T P U T

```

prompt> whatsthequiz
how many questions between 1 and 10: 5

type answer after the question
what's the capital of South Dakota :  pierre
Excellent!, well done

type answer after the question
  191
+  102
-----
   293
Excellent!, well done

type answer after the question
what artist made No Jacket Required :  who knows
I'm sorry, the answer is Phil Collins

type answer after the question
  157
+  146
-----
   303
Excellent!, well done

type answer after the question
what artist made Terrapin Station :  grateful dead
Excellent!, well done

```

We use `PromptlnRange` instead of `PromptRange` because we're using `getline` in `GiveQuestion` instead of operator `>>` (see Program Tip 9.3.) The

⁶The entire program is not shown here, but is available with the code that comes with this book.

660

Chapter 13 Inheritance for Object-Oriented Design

vector question is now a vector of pointers to Question objects instead of MathQuestion objects. Class WhatsTheQuestion is declared in *whatstheface.h*.

Program 13.7 *whatstheface.h*

```
#ifndef _WHATSTHEQUESTION_H
#define _WHATSTHEQUESTION_H

#include <string>
using namespace std;

// see "questface.h" for details on member functions
//
// A class for generating quiz questions like
// "What's the capital of Arkansas"
// "Who wrote Neuromancer"
// "What artist recorded 'Are You Gonna Go My Way'"
//
// A file of questions is read, one is used at random each time Create
// is called. The file is in the format
//
// question
// answer
// question
// answer
//
// i.e., a question uses two lines, the answer is the second line, the
// question is the first line:
//
// Terrapin Station
// Grateful Dead
// Hoist
// Phish
// It's A Shame About Ray
// Lemonheads
// -----
//
// The constructor and method Open take a prompt and a file of questions
// as parameters, e.g.,
// WhatsTheQuestion capitals("What's the capital of", "capitals.dat");

#include "questface.h"
#include "tvector.h"

class WhatsTheQuestion : public Question
{
public:
    WhatsTheQuestion();
    WhatsTheQuestion(const string& prompt,
                     const string& filename);

    virtual bool IsCorrect(const string& answer) const;
    virtual string Answer() const;
};
```

```

virtual void    Ask()                                const;

virtual void    Create();
virtual void    Open(const string& prompt,
                    const string& filename);

protected:
    struct Quest
    {
        string first;
        string second;
        Quest() {} // need vector of Quests
        Quest(const string& f, const string& s)
            : first(f),
              second(s)
        {}
    };
    tvector<Quest> myQuestions; // list of questions read
    string         myPrompt;    // prompt the user, "what's the ..."
    int            myQIndex;    // current question (index in myQuestions)
};

#endif

```

whatstheface.h

13.2.2 When Is a Method `virtual`?

We've discussed the advantages of using an inheritance hierarchy and saw how few modifications were needed in a client program like *inheritquiz.cpp*, Program 13.2, to use completely different kinds of questions. A case has been made to design inheritance hierarchies by deriving from abstract classes, but when should functions be virtual and when should they be pure virtual? One easy answer is that if you're designing an inheritance hierarchy, you should make every member function virtual.

ProgramTip 13.7: Make all methods in a superclass virtual methods. The superclass may be an abstract class in which at least some of the methods are pure virtual. It's an easy decision to make all methods in a superclass virtual. The cost is a possible mild performance penalty since virtual functions are slightly more expensive to call than nonvirtual functions. However, until you know where your code needs performance tuning, do not try to anticipate performance problems by making methods in a superclass nonvirtual.

Three classes in *inheritdemo.cpp*, Program 13.8, form a small inheritance hierarchy. We'll use the superclass `Person` and subclasses `Simpleton` and `Thinker` to demonstrate what can happen when functions in a superclass aren't virtual. In the listing and first run of the program, all classes are virtual. The class `Person` is abstract since `Person::ThinkAloud` is a pure virtual method. Implementations are provided for

662

Chapter 13 Inheritance for Object-Oriented Design

the other methods in Person, but all methods are virtual, so they can be overridden in derived classes.

Program 13.8 inheritdemo.cpp

```
#include <iostream>
#include <string>
using namespace std;

#include "dice.h"

class Person      // abstract base class for every Person
{
public:
    virtual ~Person() {}

    virtual void ThinkAloud() = 0;    // makes class abstract

    virtual void Reflect() const
    {   cout << "...As I see it, ...";
    }
    virtual string Name() const
    {   return "Ethan";
    }
};

class Simpleton : public Person      // a simple thinker
{
public:
    Simpleton(const string& name);
    virtual void ThinkAloud();

    virtual void Reflect() const;
    virtual string Name() const;

private:
    string myName;
};

class Thinker : public Person      // a cogent person
{
public:
    Thinker(const string& name);
    virtual void ThinkAloud();

    virtual void Reflect() const;
    virtual string Name() const;

private:
    string myName;
    int myThoughtCount;
};
```

```
Simpleton::Simpleton(const string& name)
    : myName(name)
  // postcondition: ready to think
  { }

void Simpleton::ThinkAloud()
  // postcondition: has thought
  {
    cout << "I don't think a lot" << endl;
  }

void Simpleton::Reflect() const
  // postcondition: has reflected
  {
    Person::Reflect();
    cout << "I'm happy" << endl;
  }

string Simpleton::Name() const
  // postcondition: returns name
  {
    return myName + ", a simpleton";
  }

Thinker::Thinker(const string& name)
    : myName(name),
      myThoughtCount(0)
  // postcondition: ready to think
  { }

void Thinker::ThinkAloud()
  // postcondition: has thought
  {
    if (myThoughtCount < 1)
      { cout << "I'm thinking about thinking" << endl;
      }
    else
      { cout << "Aha! I have found the answer!" << endl;
      }
    myThoughtCount++;
  }

void Thinker::Reflect() const
  // postcondition: has reflected
  {
    cout << "I'm worried about thinking too much" << endl;
  }

string Thinker::Name() const
  // postcondition: returns name
  {
    return myName + ", a thinker";
  }

void Think(Person & p)
```

664

Chapter 13 Inheritance for Object-Oriented Design

```

// precondition: p has thought and reflected once
{
    cout << "I am " << p.Name() << endl;
    p.ThinkAloud();
    p.Reflect();
}

int main()
{
    Simpleton s ("Sam");
    Thinker t ("Terry");
    int k;
    for(k=0; k < 2; k++)
    {
        Think(s);
        cout << "--" << endl << endl;
        Think(t);
        cout << "--" << endl << endl;
    }
    return 0;
}

```

inheritdemo.cpp

Note that `Simpleton::Reflect` calls the superclass `Reflect` method by qualifying the call with the name of the superclass. All superclass methods are inherited, and can be called even when the methods are overridden in a subclass.

O U T P U T

```

prompt> inheritdemo
I am Sam, a simpleton
I don't think a lot
...As I see it, ...I'm happy
----

I am Terry, a thinker
I'm thinking about thinking
I'm worried about thinking too much
----

I am Sam, a simpleton
I don't think a lot
...As I see it, ...I'm happy
----

I am Terry, a thinker
Aha! I have found the answer!
I'm worried about thinking too much
----

```

If we make `Person::Name` *non-virtual*, the behavior of the program changes. Each subclass inherits the nonvirtual `Name`, but any call to `Name` through a pointer or reference to the superclass `Person` cannot be overridden. What this means is that in the function `Think`, where the parameter is a `Person` reference, the call `p.Name()` will call `Person::Name` *regardless* of the type of the argument passed to `Think`. Recall that nonvirtual functions are resolved at compile-time. This means that the determination of what function is called by `p.Name()` is made when the program is compiled, not when the program is run. The determination of what function is actually called by `p.ThinkAloud()` and `p.Reflect()` is made at runtime because these functions are virtual. Because of this **late binding** of the virtual function actually called, the execution reflects arguments passed to the function at runtime rather than what the compiler can determine when the program is compiled. In the sample run that follows only one round of thinking and reflecting is shown since this is enough to see the effects of the nonvirtual function `Person::Name`: every person in the program prints the name `Ethan` although that's not really any person's name!

O U T P U T

```
prompt> inheritdemo
I am Ethan
I don't think a lot
...As I see it, ...I'm happy
----

I am Ethan
I'm thinking about thinking
I'm worried about thinking too much
----
```

Virtual Destructors. Although it hasn't mattered in the examples we've studied so far, the destructor in any class with virtual methods *must* be virtual. Many compilers issue warnings if the destructor in a class is not virtual when some other method is virtual. As we noted in Program Tip 13.4, each subclass automatically calls the superclass constructor. The same holds for subclass destructors. Whenever a subclass destructor is called, the superclass destructors will be called as well. Superclass destructor calls, like all destructor calls, are automatic. This means you must implement a superclass destructor, even for abstract classes! Although abstract classes cannot be constructed, it's very likely that you'll call a destructor through a superclass pointer. The last loop in `main` of `inheritquiz.cpp`, Program 13.2, calls a destructor through a pointer to the superclass `MathQuestion`. The destructor should be virtual to ensure that the real destructor, the one associated with the actual object being destroyed, is called. This is illustrated in Figure 13.3 where the `Subclass` destructor is called through `s`, a

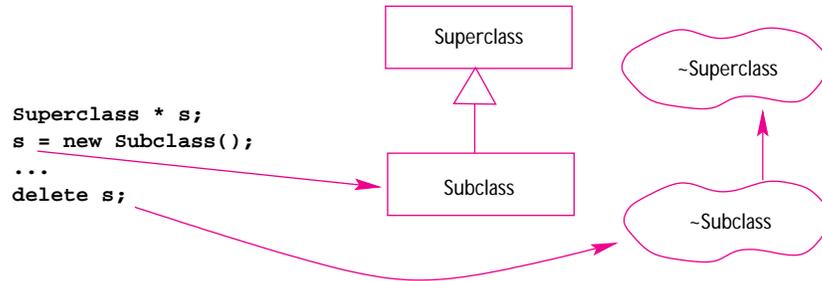


Figure 13.3 The subclass destructor automatically calls the superclass destructor, even when the superclass is abstract. The superclass must provide a destructor implementation even when the destructor is pure virtual.

Superclass pointer. The destructor `~Superclass()` is automatically called by `~Subclass()`.

When to Make a Virtual Function Pure. Any class with one pure virtual method is abstract. Every method that's an intrinsic part of a superclass interface, and that must be implemented in every subclass, should be pure virtual in the superclass. If you can decide at design time that a default implementation of a method in the superclass is a good idea, then the method can be virtual rather than pure virtual. A pure virtual method doesn't have a reasonable default implementation, but is clearly part of the interface in an inheritance hierarchy.

It's possible that you'll want default implementations for every method, but still want an abstract class. You can do this by making the destructor pure virtual, and still provide an empty-body implementation. This is a two-step process:

- Declare the destructor pure virtual, that is,

```
virtual ~Superclass() = 0;
```

- Provide an empty-body implementation in a .cpp file that must be linked in creating the final program.

```
Superclass::~~Superclass()
{
}
```

You'll get a link error if you fail to provide an implementation. Remember that a pure virtual function must be overridden, but you can provide an implementation (that can be called by subclass implementations).

Pause to Reflect



13.8 In the run of *whatsthequiz.cpp* in Section 13.2.1, the user types *pierre* as the capital of South Dakota, and the answer is acknowledged as correct. However, the entry for South Dakota in the data file *statequiz.dat* is *Pierre*, with a capital 'P.' What method judges the lowercase version *pierre* as correct? Given the declaration *whatstheface.h*, Program 13.7, write the method.

13.9 If the user had typed *Bismark* for the capital of South Dakota, what would have been printed as the correct answer. In particular, what case would be used for the first letter of the answer?

13.10 Suggest an alternative design to *WhatstheQuestion* that would enable the user to construct an instance of the class by giving the filename, but without giving the prompt used for the question. (Hint: what information is stored in the file?)

13.11 There is no destructor `~WhatstheQuestion()` declared in *whatstheface.h*, Program 13.7 nor is there a destructor declared in *mathquestface.h*, Program 13.3. However, the final loop that deletes objects through pointers in *inheritquiz.cpp* and its modification that uses class *Question* doesn't generate errors. Why?

13.12 If *Question* is modified to have a *Description* function similar to the function used by *MathQuestion* classes, but so that subclasses *must* override *Description*, then what does the declaration look like in *questface.h*, Program 13.5?

13.13 Should the function `Person::Name()` in Program 13.8 have a default implementation or would it be better to make the function pure virtual? Why?

13.14 Suppose an implementation of `Person::ThinkAloud()` is defined as follows (inline in the class declaration).

```
virtual void ThinkAloud()
{
    cout << "My brain says...";
}
```

Show how this function can be called by `Thinker::ThinkAloud()` only in the case that *Thinker* prints the message below:

Aha! I have found the answer!

13.15 If the default implementation from the previous problem is provided, and the word `virtual` removed from the declaration of `Person::ThinkAloud`, how does the output of the program change (show your answer by modifying the full run of the program).

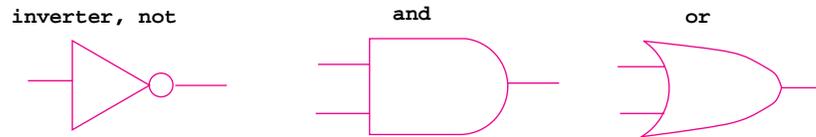


Figure 13.4 Three basic gates for building circuits.

13.3 Advanced Case Study: Gates, Circuits, and Design Patterns

In this section we'll study the design and implementation of a class hierarchy and set of programs for simulating **logic gates** and circuits.⁷ We'll see how a well-designed class hierarchy enables us to model in software the same modular, component-based circuit construction that hardware designers use. We'll use the heuristics for programming with inheritance discussed earlier in the chapter as well as some of the **design patterns** from [GHJ95] that have become part of the standard tool kit for object-oriented programmers.

13.3.1 An Introduction to Gates and Circuits

In both an abstract and concrete sense, current computers are built from gates and chips, sometimes called **digital logic**. Although computers now have specially designed chips for tasks like playing sound, doing graphics, and reading disks, at some low level everything can be built from gates that regulate when and how electricity flows through a circuit. We'll use a standard set of three gates to construct different circuits and then use these circuits to construct other circuits. Eventually this process can lead to a working computer. Instead of physically building the gates we'll model them in software. The relationship between mathematical logic and digital logic was first recognized by Claude Shannon (see his biography in Section 4.3.3.)

The three gates we'll use are shown in Figure 13.4. They are the **and-gate**, the **or-gate**, and the **inverter** or **not-gate**. Each of these gates corresponds to a boolean operator with the same name in C++. Traditionally, the behavior of these gates is shown with truth tables identical to those in Table 4.3 for the logical operators. Program 13.9 simply creates one of each gate and tests it with all possible inputs to show how gate behavior is the same as the behavior of the logical operators shown in Table 4.3.

⁷This example was motivated by a related example in [AS96]. If you read only one (other) book in computer science, that should be the one. It is simply the best introductory book on computer science and programming there is, though it's not easy reading.

Program 13.9 `gatetester.cpp`

```
#include <iostream>
using namespace std;

#include "gates.h"
#include "wires.h"

// show truth tables for each digital logic gate

int main()
{
    Gate * andg = new AndGate();
    Gate * org  = new OrGate();
    Gate * inv  = new Inverter();

    GateTester::Test(andg);
    GateTester::Test(org);
    GateTester::Test(inv);

    return 0;
}
```

`gatetester.cpp`

O U T P U T

```
prompt> gatetester
testing and (0)
-----
0 0      :      0
1 0      :      0
0 1      :      0
1 1      :      1
-----
testing or (0)
-----
0 0      :      0
1 0      :      1
0 1      :      1
1 1      :      1
-----
testing inv (0)
-----
0        :      1
1        :      0
-----
```

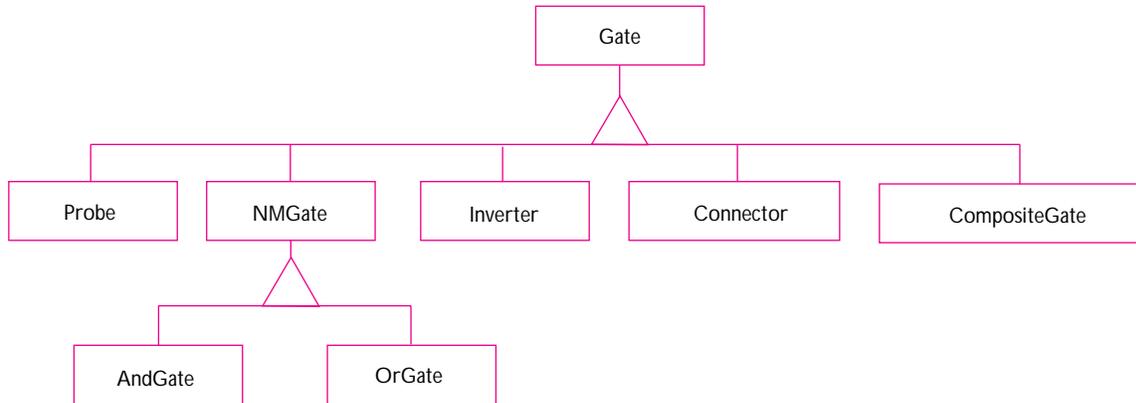


Figure 13.5 Hierarchy of components for building circuits.

The output is displayed using ones and zeros instead of true and false, but one corresponds to true and zero corresponds to false. If you look at *gatetester.cpp* carefully, you'll notice that `new` is called for three different types, but the returned pointer is assigned to variables of the same type: `Gate`. The inheritance hierarchy that enables this assignment is shown in Figure 13.5. The class `GateTester`, included via `#include "gates.h"`, contains a static method `Test`. We could have made `Test` a free function, but by making it a static function in the `GateTester` class we avoid possible name clashes with other functions named `Test`.⁸ Gates by themselves aren't very interesting; to build circuits we need to connect the gates together using wires. Complex circuits are built by combining gates and wires together. Once a circuit is built, it can become a component in other circuits, acting essentially like a more complex gate.

13.3.2 Wires, Gates, and Probes

Program 13.10, *gatewiredemo.cpp* shows another method of gate construction. Wires are created, and then gates are constructed with wires attached to each gate's input(s) and output(s). The gates in Program 13.9 were constructed without wires attached to the inputs and outputs, but as we'll see later it's possible to attach wires after a gate has been constructed as well as to construct a gate from wires as shown in Program 13.10. All three of the principle logic gates (and, or, inverter) can be given a name when constructed as shown for the `andg` gate pointer.

The gates in *gatewiredemo.cpp* are wired together as shown in Figure 13.6. An and-gate and an or-gate are attached so that the output of the and-gate feeds into the or-gate. In addition, `Probe` objects are attached to the output wires of the gates. As

⁸The C++ `namespace` feature (see Section A.2.3) could also be used to avoid name conflicts, but several compilers still don't support namespaces.

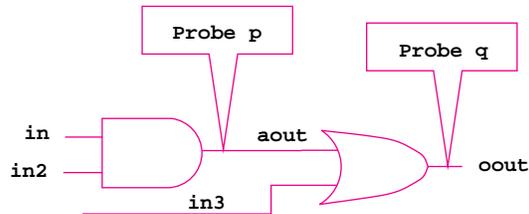


Figure 13.6 A simple example using gates, wires, and probes.

shown in Figure 13.5, a Probe *is-a* Gate. Abstractly, gates are attached to wires (and vice versa), so a probe is similar to an and-gate in this respect. A Probe object prints a message whenever the current on the wire it's monitoring changes, but also prints a message when it's first attached to a wire.

When a wire is constructed, the current on the wire is set to zero/false. The current changes when it's either explicitly changed using `Wire::SetCurrent`, or when a change in current propagates from an input wire to an output wire through a gate. A careful reading of the program and output shows that wires can be printed, and that each wire is numbered in the order in which it's created (a static counter in `wires.h`, Program G.15, keeps track of how many wires have been created). After the circuit is constructed, the probes detect and print changes caused by changes in the circuit.



Program 13.10 `gatewiredemo.cpp`

```
#include <iostream>
using namespace std;

#include "gates.h"
#include "wires.h"

int main()
{
    Wire * in  = new Wire(); // and-gate in
    Wire * in2 = new Wire(); // and-gate in
    Wire * in3 = new Wire(); // or-gate in
    Wire * aout = new Wire(); // and-gate out
    Wire * oout = new Wire(); // or-gate out

    Gate * andg = new AndGate(in,in2,aout,"andgate");
    Gate * org  = new OrGate(in3,andg->OutWire(0),oout);
    cout << "attaching probes" << endl;
    Probe * p = new Probe(aout); // attach to the and-out wire
    Probe * q = new Probe(oout); // attach to the or-out wire

    cout << "set " << *in << " on" << endl;
    in->SetSignal(true);
}
```

672

Chapter 13 Inheritance for Object-Oriented Design

```

cout << "set " << *in2 << " on" << endl;
in2->SetSignal(true);
cout << "set " << *in << " off" << endl;
in->SetSignal(false);
cout << "set " << *in3 << " on" << endl;
in3->SetSignal(true);
return 0;
}

```

 gatewiredemo.cpp

After the probes are attached, the current on wire 0, one of the and-gate inputs, is turned on (or set). Since the other and-gate input has no current, no current flows out of the and-gate. When the current to wire 1 is set, the and-gate output (wire 3) becomes set and the probe detects this. Since the and-gate output is one of the or-gate inputs, the or-gate output (wire4) is also set and the other probe detects this change. The probes continue to detect changes as current is turned off and on as illustrated in the program and output.

O U T P U T

```

prompt> gatewiredemo
attaching probes
(wire 3)      signal= 0
(wire 4)      signal= 0
set (wire 0) on
set (wire 1) on
(wire 4)      signal= 1
(wire 3)      signal= 1
set (wire 0) off
(wire 4)      signal= 0
(wire 3)      signal= 0
set (wire 2) on
(wire 4)      signal= 1

```

13.3.3 Composite Gates and Connectors

Program 13.12 shows two ways of constructing the **xor-gate** illustrated in Figure 13.7. The output of an xor-gate is set when one of its inputs is set, but not when both are set. The truth-table generated by `GateTester::Test` for an xor-gate is shown in the output. Both methods create a `CompositeGate` object, another of the gates in the hierarchy shown in Figure 13.5. A `CompositeGate` is a gate as shown in the diagram, but it's a gate made up of other gates. In particular, a composite gate can be formed from the basic gate building-blocks, but also from other composite gates. A collection of connected gates is also known as a **circuit**, so a `CompositeGate` object represents a circuit. The key idea here is that a circuit is also a gate for building other circuits.

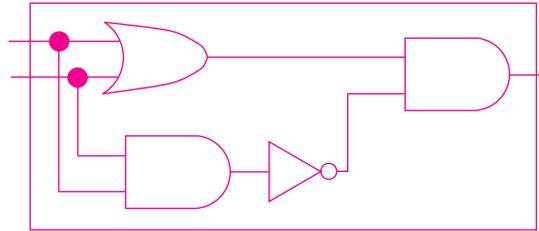


Figure 13.7 Building an xor circuit.

Program Tip 13.8: The class `CompositeGate` is a concrete example of the *Composite* design pattern [GHJ95]. There, the pattern is a solution to a problem stated as “you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects ... uniformly.”

A linked-list can also be viewed as a composite. A node is an individual item in a list, but it also represents a complete list since the node provides access to the rest of the list. Just as a node contains information and a pointer to another node, a `CompositeGate` contains information about the gate and many pointers to other gates, the gates that make up the circuit. Rather than dwell on the implementation of the class, we’ll see how it’s used to create complex circuits.

It’s almost simpler to create a new class `XorGate` than to build a composite gate that works like an xor-gate. However, creating a new class requires writing code and recompiling a program. As we’ll see in the final program from this chapter, it’s possible to create a gate-construction language or program that can build new gates while a program is running. The only member function of a class `XorGate` that differs in a substantive way from either `AndGate` or `OrGate` is `XorGate::Act`, the method that determines how a signal propagates through the gate.

Program 13.11 `xorgate.cpp`

```
class XorGate : public Gate
{
public:
    virtual void Act()
    {
        myOuts[0]->SetSignal(
            (myIns[0]->GetSignal() || myIns[1]->GetSignal()) &&
            !(myIns[0]->GetSignal() && myIns[1]->GetSignal())
        )
    }
};
```

`xorgate.cpp`

The partial class declaration and definition shown above captures in boolean logic and code exactly the relationship shown in digital logic in Figure 13.7. The output is set when either input is set, but not when both inputs are set.

Constructing CompositeGate Objects. Three `CompositeGate` methods allow a complex gate to be constructed from other gates.

- `AddGate` adds a gate to a composite gate. Presumably the added gates will be connected in some way (otherwise the composite gate won't be very useful.)
- `AddIn` adds an input wire to a composite gate. Presumably each input wire is connected to a gate that's part of the composite object. Each call of `AddIn` adds a new input wire.
- `AddOut` adds an output wire to a composite gate. As with `AddIn`, presumably each added output wire is connected to one of the gates added to the composite.

Each of these methods is shown in `MakeXOR` of Program 13.12. Note that each call of `AddIn` and `AddOut` adds a wire that is an input (respectively output) of a gate already added to the composite. The input and output wires could be specified first, then the gates added; the net effect is the same.

Using the Method `Gate::clone`. The `MakeXOR` function also shows the method `Gate::clone` applied to the `AndGate` object `ag`. The method `clone` is abstract⁹ in `Gate` so every concrete subclass must provide an implementation. Client programs typically define objects and reference them through pointers of type `Gate *` more often than by pointers of a specific subclass like `AndGate *` or `CompositeGate *`. Since `clone` is virtual, the object actually cloned returns a copy of itself.

```
void DoStuff(Gate * g)
// post: do something with a copy of g
{
    Gate * copy = g->clone();
    // what kind of gate is copy? we can't tell but
    // we can apply any generic Gate method to copy
}
```

In this example, the object referenced by `copy` is some kind of gate, and if `clone` works as expected `copy` is a duplicate of the gate `g` passed to the function `DoStuff`. The `Gate::clone` method is an example of what's often called a **virtual constructor**. The `clone` method is used to create objects, like a constructor, but the `clone` method is virtual so it creates an object whose type isn't known at compile time.

⁹Recall that I use *abstract* rather than the more C++ specific term *pure virtual*.

Program Tip 13.9: The clone method is a concrete example of what's called the *Factory Method* design pattern in [GHJ95]. There, the pattern is a solution to a problem paraphrased as “client code can't anticipate what kind of objects it must create or wants to delegate responsibility of creation to subclasses in a class hierarchy.”

Using Connectors. The functions `MakeXOR` and `MakeXOR2` illustrate the differences between calling `Connect` to connect wires to gate inputs and output (in `MakeXOR`) and constructing gates from existing wires (`MakeXOR2`). When gates are constructed without wires attached as they are in `MakeXOR`, the gate functions `InWire` and `OutWire` are used to access input wires and output wires, respectively, for attaching these wires to other wires using connectors. A connector is a gate that simply transfers current from one wire to another as though the wires are joined or soldered together.

As the output shows, the circuit created by `MakeXOR` uses more wires than the circuit created by `MakeXOR2`. When gates are constructed without wires in client code, each gate creates its own wires for input and output. Counting the input and output wires for each gate in Figure 13.7 shows that there are 11 wires: $3 \times (2 \text{ and-gates}) + 3 \times (1 \text{ or-gate}) + 1 \times (2 \text{ inverters})$. The wires for the gate created by `MakeXOR2` are explicitly created in the client program. There are fewer wires since, for example, the connections between the inputs of the rightmost and-gate (whose output is the circuit's output) and their sources (the outputs of the or-gate and inverter) require only two wires whereas four wires are used by `MakeXOR`.

Program 13.12 `xordemo.cpp`

```
#include <iostream>
using namespace std;

#include "gates.h"
#include "wires.h"
#include "tvector.h"

// illustrate connecting wires to gates using Connect

CompositeGate * MakeXOR()
// post: return an xor-gate
{
    CompositeGate * xorg = new CompositeGate(); // holds xor-gate
    Gate * ag = new AndGate(); // build components
    Gate * ag2= ag->clone(); // and gate a different way
    Gate * og = new OrGate();
    Gate * inv = new Inverter();

    Connect(og->InWire(0), ag->InWire(1) ); // wire components
    Connect(og->InWire(1), ag->InWire(0) );
    Connect(ag->OutWire(0), inv->InWire(0));
}
```

676 Chapter 13 Inheritance for Object-Oriented Design

```

Connect(inv->OutWire(0), ag2->InWire(1));
Connect(og->OutWire(0), ag2->InWire(0));

xorg->AddGate(ag); xorg->AddGate(ag2); // add gates to xor-circuit
xorg->AddGate(inv); xorg->AddGate(og);

xorg->AddOut(ag2->OutWire(0)); // add inputs/outputs
xorg->AddIn(og->InWire(0)); xorg->AddIn(og->InWire(1));

return xorg;
}

CompositeGate * MakeXOR2()
// post: returns an xor-gate
{
    CompositeGate * xorg = new CompositeGate();
    tvector<Wire *> w(6); // need 6 wires to make circuit
    tvector<Gate *> gates; // holds the gates in the xor-circuit
    int k;
    for(k=0; k < 6; k++)
    { w[k] = new Wire();
    }
    gates.push_back(new OrGate( w[0], w[1], w[2] )); // create wired gates
    gates.push_back(new AndGate(w[0], w[1], w[3] )); // share inputs
    gates.push_back(new Inverter(w[3], w[4] )); // and out->inv in
    gates.push_back(new AndGate(w[2], w[4], w[5] )); // combine or, inv

    for(k=0; k < gates.size();k++) // add gates to xor
    { xorg->AddGate(gates[k]);
    }
    xorg->AddIn(w[0]); xorg->AddIn(w[1]); // add inputs/outputs
    xorg->AddOut(w[5]);

    return xorg;
}

int main()
{
    CompositeGate * g = MakeXOR();
    CompositeGate *g2 = MakeXOR2();
    cout << "circuit has " << g->CountWires() << " wires" << endl;
    GateTester::Test(g);
    cout << "circuit has " << g2->CountWires() << " wires" << endl;
    GateTester::Test(g2);

    return 0;
}

```

xordemo.cpp

The code in `MakeXOR2` exploits the `Gate` class hierarchy by creating a vector of pointers to `Gate *` objects, but creating different kinds of gates for each pointer to reference. A vector of `Gate *` pointers is also used in the private section of the `CompositeGate` class to store the gates used in constructing the composite object. Although the functions `MakeXOR` and `MakeXOR2` create different digital circuits, the circuits are identical from

a logical view point: they compute the same logical operator as shown by the truth tables. The different functions create a `CompositeGate` using the same process.

1. Create an initially empty composite.
2. Construct gates, wire them together, and add the gates to the composite.
3. Specify input wires and output wires for the composite.
4. The composite object is finished.

As we've noted, steps two and three can be interchanged, the relative order in which these steps are executed does not affect the final composite gate.

```

                                O U T P U T
prompt> xordemo
circuit has 11 wires
testing composite: 4 gates, 2 in wires, 1 out wires
-----
0 0      :      0
1 0      :      1
0 1      :      1
1 1      :      0
-----
circuit has 6 wires
testing composite: 4 gates, 2 in wires, 1 out wires
-----
0 0      :      0
1 0      :      1
0 1      :      1
1 1      :      0
-----

```

Pause to Reflect



13.16 Suppose a probe `pin` is added to the input wire `in` as part of Program 13.10:

```

Probe * q = new Probe(out); // in original program
Probe * pin = new Probe(in); // added here

```

As a result of adding this probe three lines of output are added. What are the lines and where do they appear in the output? (Hint: one line is printed when the probe is attached.)

13.17 If the `AndGate` instance `andg` in Program 13.10 is tested at the end of `main`, the truth table printed is the standard truth table for an and-gate.



This happens even though the output of `andg` is connected to the input of the or-gate `org`. Why? (Hint: is the circuit consisting of the and-gate and or-gate combined into a `Gate` object?)

```
GateTester::Test( andg );
```

13.18 The probe `p` can be removed from the wire `about` at the end of Program 13.10 using a `Wire` member function. What's the function and what call uses it to remove the probe (see `wires.h`, Program G.15 for `Wire` methods)?

13.19 Write the function `RemoveProbe` whose header follows. (See `wires.h` and `gates.h` in How to G.)

```
void RemoveProbe(Probe * p)
// post: p is removed from the wire it monitors/probes
```

13.20 The return type of the function `MakeXOR` is `CompositeGate *` in Program 13.12, `xordemo.cpp`. If the return type is changed to `Gate *` an xor-gate is still returned, but the call of `MakeXOR` below fails to compile.

```
CompositeGate * g = MakeXOR();
```

If `g`'s type is changed to `Gate *` the definition of `g` compiles, but then the output statement below fails to compile.

```
cout << "circuit has " << g->CountWires()
      << " wires" << endl;
```

What's the cause of this behavior (hint: `CountWires` is not a `Gate` method.)

13.21 The circuit diagrammed in Figure 13.8 shows a circuit that is logically equivalent to an or-gate, but which is constructed from an and-gate and three inverters. Write a function `MakeOR` that returns a `CompositeGate` representing the circuit diagrammed in Figure 13.8. Draw a similar circuit that's logically equivalent to an and-gate using only inverters and or-gates.

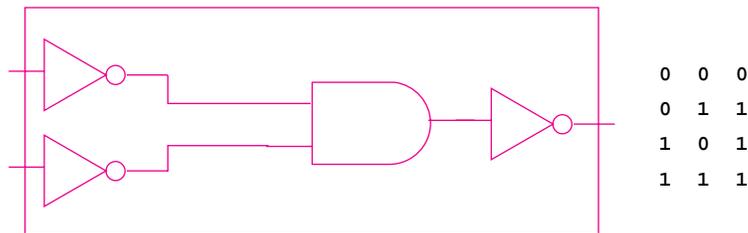


Figure 13.8 Building an or-gate from other basic gates.

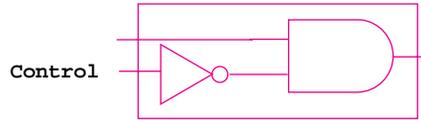


Figure 13.9 A disabler circuit.

13.22 The circuit diagrammed in Figure 13.9 is a **disabler** circuit. The signal on the wire labelled *Control* determines if the signal on the (other) input wire is passed through to the output wire of the disabler. When the control signal is zero (off), the input signal goes through to the output, (i.e., the input and the output are the same). When the control signal is set, (i.e., true/one), the input signal is stopped, or disabled, and the output wire is false/zero regardless of the value on the input wire.

Write a function `MakeDisabler` that returns a disabler circuit. Construct both gates without wires so that you must use `Connect` to wire the circuit together. How many wires are used in the circuit? (Do this exercise on paper, not necessarily by writing and testing a function.) Implement an alternative version called `MakeDisabler2` which does not use `Connect` so that both gates in the circuit are constructed with wires. How many wires are used in the circuit?

13.23 Write the method `Disabler::Act` that represents the logic of a disabler circuit. Model the function on the version of `XorGate::Act` shown in Section 13.3.3.

13.24 The **comparator** circuit shown in Figure 13.10 determines whether the signal on the wire labeled *R* is less than the signal on the wire labeled *C*, where one/zero are used for true/false. (*continued*)

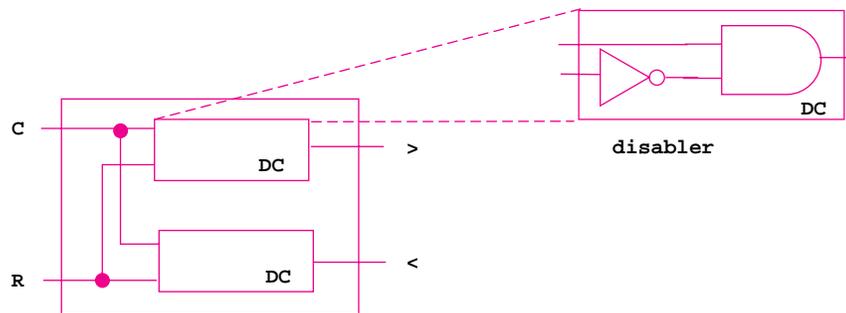


Figure 13.10 A comparator circuit for selecting the larger of two values.

Write a truth table for the circuit by tracing all four possible combinations of zero/one for inputs and labeling the corresponding outputs. Verify that if the signals are the same, the outputs are both zero. If $R < C$ then the lower output wire labeled $<$ is one/true and the upper wire is zero/false. If $R > C$ then the upper output labeled $>$ is one/true and the lower wire is zero/false.

13.25 Assume a function `MakeDisabler` is written that returns a disabler-gate/circuit. Use this function to write a `MakeComparator` function that returns a composite-gate encapsulating a comparator circuit.

13.26 Do you expect the truth tables printed by the two calls of `GateTester::Test` that follow to be the same? Why?

```
void TruthTwice(Gate * g)
{
    Gate * copy = g->clone();
    GateTester::Test(g);
    GateTester::Test(copy);
}
```

13.3.4 Implementation of the wire and Gate Classes

The interactions between classes in the `Gate` hierarchy and the class `Wire` are fairly complex. It's not essential to understand these interactions to write simple programs like the ones we studied in previous sections, but a solid understanding of the interactions is needed before you write your own `Gate` subclasses or write more involved programs.

Once we've looked at the classes and their implementations in more detail, we'll be able to make judgments about the overall design of the `Gate/Wire` framework. We'll see that there are some problems in the `Gate` class hierarchy that make it more difficult to add an `XorGate` subclass than it should be. It's not difficult to add such a class, but the process would be considerably more simple with the introduction of a new class encapsulating behavior common to `AndGate`, `OrGate`, and what would be `XorGate`. As we've stressed, software should be grown: the design process does not finish when you have a working program or prototype. Since programs and classes evolve, it makes sense to step back and examine a design and implementation after the initial kinks have been ironed out.

Program Tip 13.10: Class methods sometimes need to be refactored into other classes, or into new classes that weren't part of an initial design.

Refactoring means you don't add new functionality, but you redistribute (to existing classes) or reassign (to new classes) existing behavior and functionality to make classes and code more reusable.

As a start towards understanding the design we'll consider the simple code in Program 13.13 that creates an or-gate, attaches a probe to the output of the gate, and sets

one of the input gates to true. The interactions and method calls made by all classes for the three lines of code in *gwinteraction.cpp* are shown in Figure 13.11.

Program 13.13 *gwinteraction.cpp*

```
#include <iostream>
using namespace std;

#include "gates.h"
#include "wires.h"

int main()
{
    Gate * org = new OrGate();
    Probe * p = new Probe(org->OutWire(0));
    org->InWire(0)->SetSignal(true);

    return 0;
}
```

gwinteraction.cpp

O U T P U T

```
prompt> gwinteraction
(wire 2)      signal= 0
(wire 2)      signal= 1
```

Two separate concepts generate almost all the interactions shown in Figure 13.11. We'll give an overview of each concept, discuss why they're used in the *Wire/Gate* framework, and then provide a more in-depth look at each of them.

1. A *Wire* object can have any number of gates attached to it. Every time the signal on a wire changes, the wire notifies all the attached gates that the signal has changed using the method `Gate::Act`. Each gate responds differently when it's acted on, for example, probes print a value, or-gates propagate a true value to their output wire if one of their input wires is set, and so on.
2. When a *Gate* is constructed without wires, such as in *gwinteraction.cpp* or in *MakeXOR* as opposed to *MakeXOR2* of Program 13.12, *xordemo.cpp*, the gate creates its own wires. Rather than calling `new Wire` directly, a gate requests a wire from a *WireFactory* associated with the entire *Gate* hierarchy by a static instance variable of the *Gate* class.

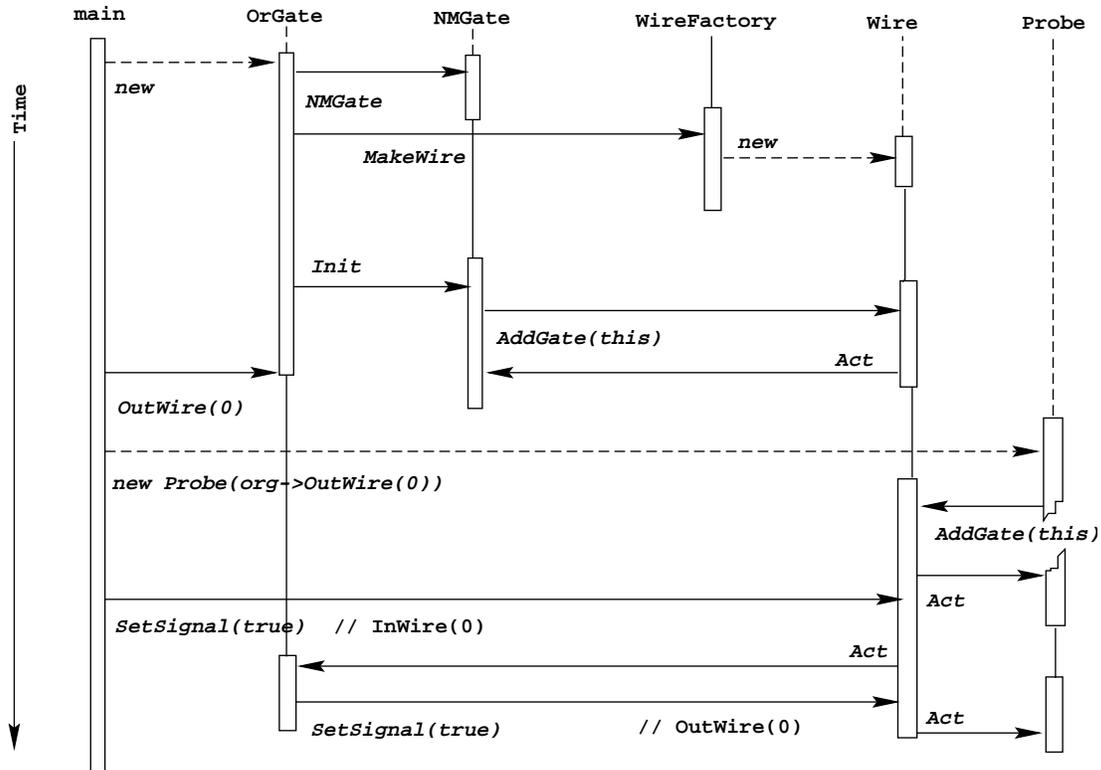


Figure 13.11 Interaction diagram: creating an or-gate with no connected wires, attaching a probe to the output of the gate, and setting the signal on the first of the gate's two inputs.

13.3.5 Gates and Wires: Observers and Observables

Look carefully at the inputs to the xor-gate diagrammed in Figure 13.7 and the comparator diagrammed in Figure 13.10. In both cases one wire is attached to the inputs of two different gates. Any change in the wire must propagate a signal through both gates. Suppose a probe is attached to one of the input wires that feeds into more than one gate. Then a change in the wire must notify two gates and a probe, which is really three gates since a probe is-a gate. How are changes in a circuit propagated? In the framework discussed here, a gate attaches itself to a wire, or can be attached by another object to a wire using the method `Wire::AddGate`. For example, a `Probe` instance adds itself during construction to the wire it probes.

```

Probe::Probe(Wire * w)
    : myWire(w)
    // post: probe is attached to w
    {
    
```

```

myWire->AddGate(this);
}

```

It's almost as though each attached gate listens to the wire, waiting for a change. However, a gate doesn't actively listen, it is notified by the wire when the wire's signal changes. The wire notifies all the gates that have been attached to it using the following code.

```

void Wire::SetSignal(bool signal)
// post: notify attached/listening gates if signal changes
{
    if (signal != mySignal)
    {
        mySignal = signal;
        int k;
        for(k=0; k < myGates.size(); k++)
        {
            myGates[k]->Act( );
        }
    }
}

```



You can look at the code in *wires.h* for details (see How to G), but the code above is mostly self-explanatory from the names of the instance variables and the syntax of how they're used — for example, *myGates* seems to be a *tvector* object from how it's used. Gates that have been attached using *AddGate* can subsequently be removed using *Wire::RemoveGate*. Gate identity for removal is based on pointer values, so any object added can be removed since the address of an object doesn't change.

Program Tip 13.11: In [GHJ95] the *Observer* pattern is a solution to a problem “when a change to one object requires changing others, and you don't know how many objects need to be changed.” The *Observer* pattern is sometimes called *Observer/Observable* or *Publish/Subscribe*.

In the code above you can see that a wire's gates are notified in the same order in which they are added to the wire. Suppose *Wire* object *w2* notifies the first of the two gates that are (hypothetically) attached to *w2*. Since a gate's *Act* method may set other wires, that will in turn call other *Act* methods; the second gate attached to *w2* may have its *Act* method invoked well after other gates have acted. In one of the modifications in the *Exercise* section you'll be asked to introduce time into the *Wire/Gate* framework to account for these anomalies.

The *Observer* pattern is common outside of programming. Volunteer firemen are notified when there's an event they must respond to, but the firemen do not actively phone the fire department to find fires. The firemen correspond to gates in our framework; the fire department is the wire notifying the firemen. Auctions sometimes model the pattern: bidders are notified when a new, higher bid has been made. A bidder actively monitoring new bids doesn't quite fit the model, but a bidder that responds only when notified of a new bid does.

Bjarne Stroustrup (b. 195?)

Bjarne Stroustrup is truly the “father” of C++. He began its design in 1979 and is still involved with both design and implementation of the language. His interests span computer science, history, and literature. In his own words:



... C++ owes as much to novelists and essayists such as Martin A. Hansen, Albert Camus, and George Orwell, who never saw a computer, as it does to computer scientists such as David Gries, Don Knuth, and Roger Needham. Often, when I was tempted to outlaw a feature I personally disliked, I refrained from doing so because I did not think I had the right to force my views on others. In writing about creating software, Stroustrup (p. 693) [Str97] mentions several things to keep in mind, three are ideas we’ve emphasized in this book: (1) There are no “cook-book” methods that can replace intelligence, experience, and good taste in design and programming, (2) Experimentation is essential for all nontrivial software development, and

(3) Design and programming are iterative activities.

Stroustrup notes that it is as difficult to define what a programming language is as to define computer science.

Is a programming language a tool for instructing machines? A means of communicating between programmers? A vehicle for expressing high-level designs? A notation for algorithms? A way of expressing relationships between concepts? A tool for experimentation? A means of controlling computerized devices? My view is that a general-purpose programming language must be all of those to serve its diverse set of users.

For his work in the design of C++, Stroustrup was awarded the 1994 ACM Grace Murray Hopper award, given for fundamental contributions made to computer science by work done before the age of 30. Most of this material is taken from [Str94].

13.3.6 Encapsulating Construction in wireFactory

The simple three-line program in Program 13.13, constructs an or-gate without providing wires when the or-gate is constructed. The or-gate makes its own wires, and the program connects a probe to the created output wire. As we saw in the two different functions

MakeXOR and MakeXOR2 of Program 13.12, a gate can be created by attaching existing wires to the gate when the gate is constructed, or by creating a gate and then connecting wires to the input/output wires the gate constructs itself. Where do these self-constructed wires come from? The simplest method is to create new wires using `new Wire()` — sample code for the `Inverter` constructor shows this (this isn't the real constructor, which uses a different technique discussed later). An `Inverter` has an input, an output, a name, and a number.

```
Inverter::Inverter(const string& name)
    : myIn(new Wire(name)), myOut(new Wire(name))
      myName(name), myNumber(ourCount)
{
    ourCount++;
    myIn->AddGate(this);
}
```

Since an `Inverter` creates the wires using the `new` operator, the class is responsible for deleting the wires in its destructor. This approach tightly couples the `Gate` and `Wire` classes. If a better wire class is designed, or we want to run a circuit simulation using a `LowEnergyWire` class representing a new kind of wire that's a subclass of `Wire`, we'll have to rewrite every gate's constructor to use the new kind of wire. We can't reduce the coupling inherent in the circuit framework because wires and gates do depend on each other, but we can reduce the coupling in how gates create wires. To do this we design a `WireFactory` class. When a client wants a wire, the wire is "ordered" from the factory rather than constructed using `new`. If a new wire class is created, we order wires from a new factory that makes the new kind of wires. Because we use inheritance to model is-a relationships, the new kind of wires can be used in place of the original wires since, for example, a `LowEnergyWire` is-a `Wire`. By isolating wire creation in a `WireFactory`, changing the kinds of wires used by all gates means simply changing the factory, and the factory is created in one place so it can be changed easily. The `Inverter` constructor actually used in `gates.cpp` illustrates how a factory isolates wire construction in one place.

```
Inverter::Inverter(const string& name)
    : myIn(ourWireFactory->MakeWire(name)),
      myOut(ourWireFactory->MakeWire(name)),
      myName(name), myNumber(ourCount)
{
    ourCount++;
    myIn->AddGate(this);
}
```

The `my/our` naming convention tells us that `ourWireFactory` is a static instance variable. The factory is shared by every `Gate` object since it's defined as a protected static data member in the abstract `Gate` superclass. This means every `Inverter`, every `AndGate`, and every gate subclass not yet implemented can share the factory.

Program Tip 13.12: Using a factory class to isolate object creation decreases the coupling between the created objects and their collaborating classes. This design pattern is called *Abstract Factory* in [GHJ95]. A factory class is used when “a system should be independent of how its products are created, composed, and represented” or when “a system should be configured with one of multiple families of products.”

Our `WireFactory` class is not abstract, but we’ll explore how to create more than one kind of factory in the exercises by creating an abstract base class from which `WireFactory` derives. The `Gate::clone` method outlined in Program Tip 13.9 as a realization of a factory *method* shares characteristics with the `WireFactory` class that is a factory *class*: both isolate object creation so that clients can use objects without knowing how to create them.

13.3.7 Refactoring: Creating a `BinaryGate` Class

When I first designed the `Gate` hierarchy in Figure 13.5 I anticipated creating classes like `And3Gate`, an and-gate with three inputs that sets its output only when all three inputs are set. I considered an `And3Gate` to be a 3-1-gate, a gate with three inputs and one output. The existing `AndGate` class represents a 2-1-gate while the comparator circuit diagrammed in Figure 13.10 is a 2-2-gate with two inputs and two outputs. Similarly, the full-adder diagrammed in Figure 13.13 is a 3-2-gate. Thinking there would be some common behavior in these gates I created a class `NMGate` to model an n-m-gate as I’ve just described. Since a subclass is responsible for calling its superclass constructor, this leads to the constructor below for an `AndGate` instance constructed without wires.

```
AndGate::AndGate(const string& name)
    : NMGate(ourCount, name)
// post: this and-gate is constructed
{
    tvector<Wire *> ins(2), outs(1);
    ins[0] = ourWireFactory->MakeWire(myName);
    ins[1] = ourWireFactory->MakeWire(myName);
    outs[0] = ourWireFactory->MakeWire(myName);
    NMGate::Init(ins, outs);
    ourCount++;
}
```

The `AndGate` constructor creates two input wires, one output wire, and puts these wires into vectors for initializing the parent `NMGate` class. The general class `NMGate` is initialized with vectors of wires for input and output so that it can be used for a 3-2-gate as well as an 8-8-gate. The `OrGate` constructor shows striking similarities to the `AndGate`.

```
OrGate::OrGate(const string& name)
    : NMGate(ourCount, name)
```

```

{
    tvector<Wire *> ins(2), outs(1);
    ins[0] = ourWireFactory->MakeWire(myName);
    ins[1] = ourWireFactory->MakeWire(myName);
    outs[0] = ourWireFactory->MakeWire(myName);
    NMGate::Init(ins,outs);
    ourCount++;
}

```

This duplicated code will be replicated in any new 2-1-gate, (e.g., if we implement an `XorGate` class). The `Act` methods of these classes differ because the gates model different logic, and the `clone` methods differ since each gate must return a copy of itself, but the other `AndGate` and `OrGate` methods are the same. Since 2-1-gates are quite common, and we may be implementing more “basic” 2-1-gates in the future, it’s probably a good idea to refactor the behavior in common to the 2-1-gates into a new class `BinaryGate`. The new class derives from `NMGate` and is a parent class to `AndGate` and `OrGate`. The `AndGate` constructor will change as follows.

```

AndGate::AndGate(const string& name)
    : BinaryGate(ourCount,name)
// post: this and-gate is constructed
{
    ourCount++;
}

```

The behavior common to the `AndGate` and `OrGate` constructors has been factored out into the `BinaryGate` constructor. Similarly, all the methods whose behavior is the same in the binary gate subclasses are factored into the new `BinaryGate` superclass.

Pause to Reflect



13.27 The method `Wire::AddGate` is implemented as follows.

```

void Wire::AddGate(Gate * g)
// post: g added to gate collection, g->Act() called
{
    myGates.push_back(g);
    g->Act();
}

```

Identify each call of `g->Act()` whose source is `AddGate` that appears in the interaction diagram of Figure 13.11. Which of the calls generate(s) output?

13.28 Constructing an `Inverter` and connecting a probe to its output generates the output shown. (*continued*)

```

Gate * inv = new Inverter();
Probe * p = new Probe(inv->OutWire(0));

```

O U T P U T

```
(wire 1)      signal= 1
```

Why is the wire labeled `(wire 1)`, where is wire 0? Draw an interaction diagram like Figure 13.11 for these two statements. Trace all method calls, particularly the `Gate::Act` calls, and show why the call of `g->Act()` in `Wire::AddGate` shown in the previous exercise is necessary to get the behavior shown in the output — what would the output of the probe be if the call `g->Act()` wasn't included in the method `AddGate`? Why?

- 13.29** The statements below construct a disabler circuit as diagrammed in Figure 13.9. The circuit isn't formed as a composite, but the gates and wires together make a disabler circuit with a probe attached to the circuit's output wire.

```
Wire * controller= new Wire();
Gate * ag=        new AndGate();
Gate * inv=       new Inverter(controller, ag->InWire(1));
Probe * p=        new Probe(ag->OutWire(0));
ag->InWire(0)->SetSignal(true); // send a signal through
```

Since the `controller` is `false/zero` when constructed, the signal set should propagate through the disabler. Draw an interaction diagram like Figure 13.11 for these five statements.

- 13.30** As implemented, the `WireFactory` class cannot recycle used wires, (i.e., if a `Gate` is destroyed, the wires it may have ordered from the factory are not reused). The factory does keep track of all the wires ever allocated/ordered, and cleans the wires up when the factory ceases to exist.

In what function does the current `WireFactory` destroy all the wires allocated during the factory's lifetime? Sketch a design that would allow the factory to recycle wires no longer needed. You'll need to identify how the factory stores recycled wires and how the factory collaborates with the `Gate` classes to get wires back when a gate no longer needs them.

- 13.31** The class `NMGate` is an abstract class because it has at least one abstract/pure virtual function, (e.g., `Act`). However, there is an `NMGate` constructor and an `NMGate` class has state: the input and output wires. Why is the class an abstract class, which means it's not possible to create an `NMGate` object, but the class still has a constructor and state? Note that the statement below will not compile for two reasons: the constructor is protected and the class is abstract.

```
Gate * g = new NMGate(); // won't compile
```

13.32 Why is `ourCount++` used in the body of the refactored `AndGate` constructor at the end of Section 13.3.7? Why isn't the increment factored into the `BinaryGate` constructor?

13.33 The following statement, added as the last statement in `main` of Program 13.12, `xordemo.cpp`, produces the output shown.

```
cout << g2->deepString() << endl;
```

The output shows the components of the composite gate `g2` created by `MakeXOR2`. The method `deepString` is implemented in each `Gate` subclasses, although it often defaults to the same function as `toString`. Why are the and gates numbered 2 and 3? Where are and gates numbered 0 and 1? Draw the circuit for this composite and label every gate and wire with its number.

OUTPUT

```
composite: 4 gates, 2 in wires, 1 out wires
all-in   (wire 11) (wire 12)
all-out  (wire 16)
      or (1)
      in  (wire 11) (wire 12)          out  (wire 13)
----
      and (2)
      in  (wire 11) (wire 12)          out  (wire 14)
----
      inv (1)
      in  (wire 14)          out (wire 15)
----
      and (3)
      in  (wire 13) (wire 15)          out  (wire 16)
----
-----
```

13.34 Instead of refactoring `AndGate` and `OrGate` into a new `BinaryGate` class, suppose a new constructor is added to the `NMGate` class in which the number of inputs and outputs is specified as shown in the following:

Is this a better solution than introducing a new class `BinaryGate`? Why? Write the constructor that takes the number of inputs and outputs as parameters.

```
AndGate::AndGate(const string& name)
    : NMGate(2,1,ourCount,name)
// post: this and-gate is constructed
{
    ourCount++;
}
```

13.3.8 Interactive Circuit Building

Program 13.12, *xordemo.cpp*, shows how a composite circuit can be built by creating gates and wires, then wiring them together. In Section 13.3.3 we described how to create new class declarations and definitions using an `XorGate` class as an example. Both these methods for creating circuits require writing, compiling, testing, and debugging programs. A different approach is outlined in the run of *circuitbuilder.cpp*. A complete version of this program is not provided; you'll be asked to write it as an exercise. We'll discuss why it's a useful program and some of the design issues that arise in developing it.

A graphical circuit-building program in which the user creates new gates by choosing from a palette of standard gates, uses the mouse to wire gates together, and tests the circuits built, might be the best way of designing and building new circuits. However, a text-base interactive circuit builder is easier to design and implement. Many of the classes and ideas in a text-based program may transfer to a graphics-based program, so we'll view the text-based program as a useful prototype.

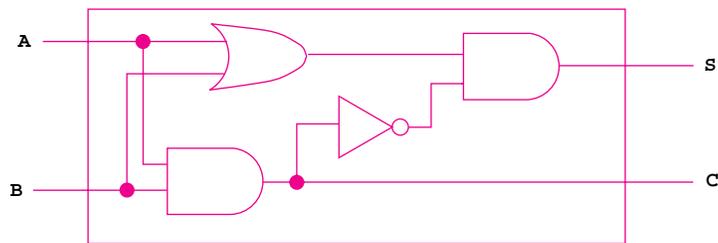


Figure 13.12 Building A half-adder circuit.

We'll use the interactive circuit building program to build a *half-adder*, a circuit for adding two one-digit binary numbers diagrammed in Figure 13.12.¹⁰ We'll use the half-adder to build a *full-adder*, a circuit that basically adds three one-bit numbers, though we'll view the inputs as two numbers and a carry from a previous addition, diagrammed

¹⁰A binary digit is usually called a **bit**, which is almost an acronym for **binary digit**.

in Figure 13.13. Full-adders can be wired together easily to form an *n-bit ripple-carry adder* for adding two *n*-bit binary numbers that we'll explore in an exercise.

Binary, or base 2, numbers are added just like base 10 numbers, but since the only values of a binary digit (or bit) are zero and one, we get Table 13.1 as a description of the half-adder.

Table 13.1 Adding two one-bit numbers

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The output labeled *S* in Figure 13.12 and Table 13.1 is the sum of two bits. The output labeled *C* is the carry. Since we have $1 + 1 = 10$ in base 2, the last line of the table shows the sum is zero and the carry is one, where the sum is the rightmost or least-significant digit. Similarly in Figure 13.13 the sum and carry represent adding the three input bits. A table for the full-adder is shown in the output of *circuitbuilder.cpp*.

Before looking at a run of the program we'll outline a list of requirements for an interactive circuit builder. The program doesn't meet all these requirements in the run shown, but you can add features as explored in chapter exercises.

1. The program should allow the user to choose standard gates for building circuits, but the list of gates should grow to include circuits built during the program. In other words, the program may start with only three gates (and, or, inverter), but any circuits built with the program become gates used in building other circuits.
2. The program should be simple to use, commands should correspond to user expectations. First-time users should be able to use the program without much help, but experienced users should be able to use their experience to build circuits quickly.
3. The program should be able to load circuits built by the program. This means the user should be able to save newly constructed circuits and load these circuits in a later run.
4. Connecting gates and wires should not require an in-depth knowledge of the `Gate` and `Wire` classes we've studied. Circuit designers shouldn't need to be experts in object-oriented programming and design to use the program.
5. The program should be flexible enough to adapt to new requirements we expect to receive from users once the program has been reviewed and tested. For example, users make mistakes in building circuits; it would be nice to support *undo* features to change gates and connections already created.

In the run below there is no facility for saving and loading circuits and there is no *undo* command, but attempts are made to meet the other requirements. The program shows an initial collection of the three standard gates available for creating circuits. In the run,

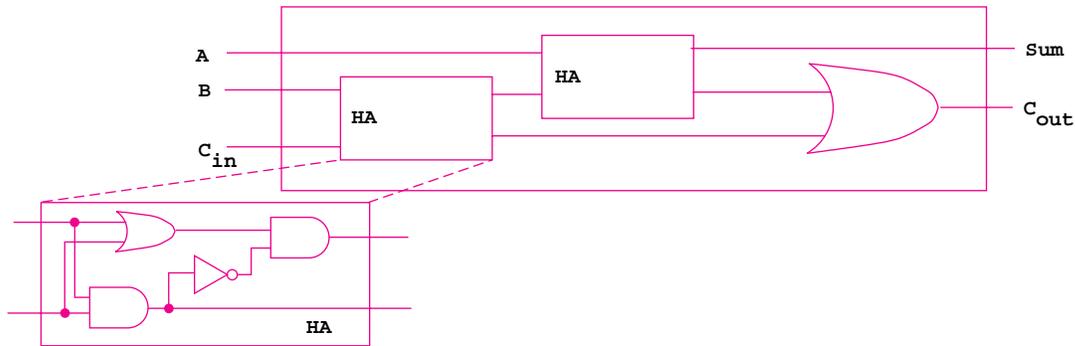


Figure 13.13 Building a full-adder from half-adders and an or-gate.

the user builds the half-adder diagrammed in Figure 13.12 by creating gates, printing the composite made from the gates in order to find the name of each wire, then connecting the gates and specifying inputs and outputs for the composite gate constructed. After the new circuit is finished, the user types *stop*, the circuit is tested, and the new circuit is added to the list of available gates.

The full-adder diagrammed in Figure 13.13 is built next using the same process:

- Gates are added to the composite: two half-adders and an or-gate.
- The composite is printed (the half-adders show up as composites), the wires between the gates are connected, and the inputs and outputs are specified.
- The circuit is finished, tested, and added to the tool kit of available circuits.

Designing for Both Novice and Expert Users. The command *add* which adds gates to the composite being constructed comes in three forms, each illustrated in the run.

- *add and*, the user specifies the gate to add
- *add*, the user doesn't specify a gate, and is prompted for one
- the user presses enter/return when prompted for a gate and list of available gates is printed (see the end of the output)

Minimal Knowledge of Gate and Wire Classes. Every gate displayed is shown with inputs and outputs. The input and output wires are numbered, and users connect wires by using a wire's number rather than typing `and(1)->InWire(0)` which might be used in a program, but shouldn't be demanded from a user¹¹.

¹¹Implementation aside: wire numbers can be used to find wires only because a `WireFactory` supports wire lookup by number.

OUTPUT

```
prompt> circuitbuilder
0.      and
1.      or
2.      inverter
command: add and
command: add and
command: add or
command: add inverter
command: show
current circuit
composite: 4 gates, 0 in wires, 0 out wires
all-in
all-out
      and (1)
      in (wire 8) (wire 9)      out (wire 10)
----
      and (2)
      in (wire 11) (wire 12)    out (wire 13)
----
      or (1)
      in (wire 14) (wire 15)    out (wire 16)
----
      inv (1)
      in (wire 17)      out (wire 18)
-----
connections: none

command: connect 10 17
command: connect 18 12
command: connect 16 11
command: connect 14 8
command: connect 9 15
command: in 14
command: in 9
command: out 13
command: out 10
command: test
output continued →
```

```

                                O U T P U T
testing composite: 4 gates, 2 in wires, 2 out wires
-----
0 0      :      0 0
1 0      :      1 0
0 1      :      1 0
1 1      :      0 1
-----
command: stop
name for circuit: half
command: add half
command: add half
command: add or
command: show
current circuit
composite: 3 gates, 0 in wires, 0 out wires
all-in
all-out
           composite: 4 gates, 2 in wires, 2 out wires
all-in   (wire 25) (wire 20)
all-out  (wire 24) (wire 21)
output elided/removed
-----
           composite: 4 gates, 2 in wires, 2 out wires
all-in   (wire 36) (wire 31)
all-out  (wire 35) (wire 32)
output elided/removed
-----
           or (4)
           in (wire 41) (wire 42)      out (wire 43)
connections: none
command: connect 24 31
command: connect 21 42
command: connect 32 41
command: in 36
command: in 25
command: in 20
command: out 35
command: out 43
command: test

output continued →

```

```

                                O U T P U T
testing composite: 3 gates, 3 in wires, 2 out wires
-----
0 0 0   :   0 0
1 0 0   :   1 0
0 1 0   :   1 0
1 1 0   :   0 1
0 0 1   :   1 0
1 0 1   :   0 1
0 1 1   :   0 1
1 1 1   :   1 1
-----
command: stop
name for circuit: full
command: add
gate name:
0.      and
1.      or
2.      inverter
3.      half
4.      full

```

13.3.9 simpleMap: Mapping Names to Gates



In my prototype for the interactive builder program I used a structure called a *map*. I'll show the simple version I used in this prototype which is good enough for the prototype, easy to understand, and not fully functional. You don't need the class `SimpleMap` to write the interactive circuit builder, but you'll need to implement something just like it (see *simplemap.h*, Program G.17.)

A `SimpleMap` is templated on two classes: one class is the **value** stored in the map, the other is the **key** used to look up a value. In *simplemapdemo.cpp*, `Gate *` values are stored in the map and `int` values are keys to retrieve gate pointers. The same kind of map is used in *circuitbuilder.cpp*, but strings are used to lookup a gate rather than integers. Users are more comfortable typing `add` and than typing `add 0`, where 0 is the index of the and-gate stored in a map.

Program 13.14 `simplemapdemo.cpp`

```

#include <iostream>
using namespace std;

```

```

#include "simplemap.h"
#include "gates.h"

int main()
{
    SimpleMap<int, Gate *> gatemap;
    gatemap.insert(0, new AndGate("map-and-gate"));
    gatemap.insert(1, new OrGate("map-or-gate"));
    gatemap.insert(2, new Inverter("map-not-gate"));

    Gate * g = 0; // get g from map
    SimpleMapIterator<int, Gate*> git(gatemap);
    for(git.Init(); git.HasMore(); git.Next())
    {   int index = git.Current();
        g = gatemap.getValue(index);
        cout << index << "\t" << *g << "\t" << *(g->clone()) << endl;
    }
    return 0;
}

```

simplemapdemo.cpp

The program shows how a map works as a gate tool kit. The program retrieves a gate and makes a copy of it using `clone`. The copy could be added to a composite being constructed by the user. When a new circuit is finished it can be easily added to the tool kit using the method `SimpleMap::insert`.

O U T P U T

```

prompt> simplemapdemo
0      and (0) map-and-gate      and (1) map-and-gate
1      or (0) map-or-gate       or (1) map-or-gate
2      inv (0) map-not-gate     inv (1) map-not-gate

```

13.4 Chapter Review

We discussed inheritance, a powerful technique used in object-oriented programming for reusing a common interface. We saw several examples of inheritance hierarchies in which superclasses specified an interface, and subclasses implemented the interface with different behavior, but using a common naming convention. Inheritance allows an object that's an instance of a subclass to be substituted for, or used-as-an instance of, the corresponding superclass. In this book inheritance always models an "is-a" relationship, which ensures that objects can be substituted for other objects up an inheritance hierarchy.

Topics covered include:

- Streams form an inheritance hierarchy. A function with an `istream` parameter can receive many kinds of streams as arguments including `cin`, `ifstream`, and

istream objects.

- Prototypes are first-attempts at designing and implementing a program or classes that allow the programmer and the client to get a better idea of where a project is headed.
- Inheritance in C++ requires superclass functions to be declared *virtual* so that subclasses can change or specialize behavior. We use *public* inheritance which models an is-a relationship. Virtual functions are also called polymorphic functions. (Other uses of inheritance are possible in C++, but we use inheritance only with virtual functions and only with public inheritance.)
- Virtual superclass functions are always virtual in subclasses, but the word `virtual` isn't required. It's a good idea to continue to identify functions as virtual even in subclasses, because a subclass may evolve into a superclass.
- Subclasses should call superclass constructors explicitly, otherwise an implicit call will be generated by the compiler.
- An inherited virtual function can be used directly, overridden completely, or overridden while still calling the inherited function using `Super::function` syntax.
- Data and functions declared as *protected* are accessible in subclasses, but not to client programs. Data and functions declared as *private* are not accessible to subclasses except using accessor and mutator functions that might be provided by the superclass. Nevertheless, a subclass contains the private data, but the data isn't directly accessible.
- Abstract base classes contain one pure virtual function, a function identified with the ugly syntax of `= 0`. An abstract base class is an interface, it's not possible to define an object whose type is an abstract base class. It's very common, however, to define objects whose type is `ABC *` where `ABC` is an abstract base class. An abstract base/superclass pointer can reference any object derived from the superclass.
- Flexible software should be extendible, programming in the future tense is a good idea. Using abstract classes that can have some default function definitions, but should have little state, is part of good programming practice.
- Several design patterns were used in designing and implementing a `Gate/Wire` framework for modeling digital circuits. The patterns used include *Composite*, *Factory*, *Abstract Factory*, and *Observer*.
- Programs should be grown rather than built; refactoring classes and functions is part of growing good software.
- A class `SimpleMap` is a usable prototype of the map classes you'll study as you continue with computer science and programming. The map class facilitates the implementation of an interactive circuit-building program.

13.5 Exercises

- 13.1** Design a hierarchy of math quiz questions that cover the operations of addition, subtraction, multiplication, and division. You might also consider questions involving

ratios, fractions, or other parts of basic mathematics. Each kind of question should have both easy and hard versions, (i.e., addition might require carrying in the hard version). Keep the classes simple to make it possible to write a complete program; assume the user is in fourth or fifth grade.

Design and implement a quiz class that uses the questions you've just designed (and tested). The quiz should use different questions, and the questions should get more difficult if the user does well. If a user isn't doing well, the questions should get simpler. The quiz class should give a quiz to one student, not to two or more students at the same time. Ideally, the quiz class should record a student's scores in a file so that the student's progress can be tracked over several runs of the program.

13.2 Implement the class `MultipleChoice` shown in Figure 13.1. You'll need to decide on some format for storing multiple choice questions in a file, and specify a file when a `MultipleChoice` question object is created. Incorporate the new question into *inheritquiz.cpp*, Program 13.2, or design a new quiz program that uses several different quiz questions.

13.3 We studied a templated class `LinkSet` designed and implemented in Section 12.3.6 (see Programs 12.11 and 12.12, the interface and implementation, respectively.) New elements were added to the front of the linked list representing the set elements. Design a class like the untemplated version of the set class, `LinkStringSet`, that was developed first. The new class supports only the operations `Add` and `Size`. Call the class `WordList`; it can be used to track the unique words in a text file as follows.

```
void ReadStream(WordStreamIterator& input,
               WordList * list)
// post: list contains one copy of each word in input
{
    string word;
    for(input.Init(); input.HasMore(); input.Next())
    {
        word = input.Current();
        ToLower(word);
        StripPunc(word);
        list->Add(word);
    }
    cout << list->Size() << " different words" << endl;
}
```

Make the function `Add` a pure virtual function and make the helper function `FindNode` from `LinkStringSet` virtual and protected rather than private. Then implement three subclasses each of which uses a different technique for maintaining the linked list (you may decide to use doubly linked lists which make the third subclass slightly simpler to implement).

- A class `AddAtFront` that adds new words to the front of the linked list. This is similar to the class `LinkStringSet`.
- A class `AddAtBack` that adds new words to the end of the linked list (keep a pointer to the last node, or use a circularly linked list).

- A class `SelfOrg` that adds new nodes at the back, but when a node is found using the virtual, protected `FindNode`, the node is moved closer to the front by one position. The idea is that words that occur many times move closer to the front of the list so that they'll be found sooner.

Test each subclass using the function `ReadStream` shown above. Time the implementations on several text files. Try to provide reasons for the timings you observe. As a challenge, make two additions to the classes once they work. (1) Add an iterator class to access the elements. The iterator class will need to be a friend of the superclass `WordList`, but friendship is *not* inherited by subclasses. You'll need to be careful in designing the hierarchy and iterator so the iterator works with any subclass. (2) Make the classes templated.

- 13.4** Program 12.4, *frogwalk3.cpp* in Section 12.1.6, shows how to attach an object that monitors two random walkers to each of the walkers. The class `Walker` is being observed by the class `WalkRecorder`, though we didn't use the term *Observer* when we discussed the example in Chapter 12.

Create an inheritance hierarchy for `WalkRecorder` objects that monitor a random walker in different ways. Walkers should accept any number of `WalkRecorders`, rather than just one, by storing a vector of pointers rather than a single pointer to a `WalkRecorder`. Implement at least two different recorders, but try to come up with other recorders that you think are interesting or useful.

- Implement a recorder that works like the original `WalkRecorder` in tracking every location of all the walkers it's recording.
- Implement an `ExtremeRecorder` class that tracks just the locations that are furthest left (lowest) and right (highest) reached by any walker being monitored by the `ExtremeRecorder`. Alternatively, have the recorder keep track of one pair of extremes per walker rather than one pair of extremes for all walkers (this is tricky).
- Use the graphics package in *How to H* and create a class that displays a walker as a moving square on a canvas. Each walker monitored should appear as a different color. Walkers can supply their own colors, or the recorder can associate a color with a walker (it could do this using a `SimpleMap` object or in several other ways).



- 13.5** Design a hierarchy of walkers each of which behave differently. The walkers should wander in two-dimensions, so a walker's location is given by a `Point` object (see *point.h*, Program G.10). The superclass for all walkers should be named `Walker`.

Design a `WalkerWorld` class that holds all the walkers. `WalkerWorld::Step` asks the world to ask each of its walkers to take one step, taking a step is a virtual function with different implementations by different `Walker` subclasses. You can consider implementing a hierarchy of `WalkerWorld` classes too, but at first the dimensions of the world in which the walkers roam should be fixed when the world is created. The lower-left corner of the world has location $(0, 0)$; the upper-right corner has location $(\text{maxX}, \text{maxY})$. In a world of size 50×100 the upper-right corner has coordinates $(49, 99)$.

Consider the following different behaviors for step-taking, but you should be imaginative in coming up with new behaviors. A walker should always start in the middle of the world.

- A random walker that steps left, right, up, and down with equal probability. A walker at the edge of the world, for example, whose location is $(0,x)$, can't move off the edge, but may have only three directions to move. A walker in the corner of the world has only two choices.
- A walker that walks immediately to the north edge of the world and then hugs the wall circling the world in a clockwise direction.
- A walker that wraps around the edge of the world, for example, if it chooses to walk left/west from location $(0,y)$ its location becomes $(\max X,y)$.

You'll probably want to add at least one `WalkRecorder` class to monitor the walkers; a graphics class makes for enjoyable viewing.

- 13.6** Function objects were used to pass comparison functions encapsulated as objects to sorting functions; see Section 11.3 for details. It's possible to use inheritance rather than templates to enforce the common interface used by the comparison function objects described in Section 11.3. Show how the function header below can be used to sort using function objects, although the function is only templated on one parameter (contrast it to the declaration for `InsertSort` in *sortall.h*, Program G.14.)



```
template <class Type>
void InsertSort(tvector<Type> & a,
int size, const Comparer & comp);
```

You should show how to define an abstract `Comparer` class, and how to derive subclasses that are used to sort by different criteria.

- 13.7** The circuit constructed by the statements below is self-referential. Draw the circuit and trace the calls of `Gate::Act` through the or-gate, inverter, and probe. What happens if the circuit is programmed? What happens if the or-gate is changed to an and-gate?

```
Gate * org = new OrGate("trouble");
Gate * inv = new Inverter();
Probe * p = new Probe(inv->OutWire(0));

Connect(org->OutWire(0), inv->InWire(0));
Connect(inv->OutWire(0), org->InWire(1))
```

- 13.8** Implement a complete program for interactively building circuits. Invent a circuit-description language you can use to write circuits to files and read them back. You should try to use a `Factory` for creating the gates and circuits used in the program, but you'll need a factory to which you can add new circuits created while the program is running. Using a `SimpleMap` can make the factory implementation easier, but you'll need to think very carefully about how to design the program.
- 13.9** Implement a class `GateFactory` that encapsulates creation of the four standard gate classes: `AndGate`, `OrGate`, `Inverter`, `CompositeGate` as well as a class

XorGate. The factory class is used like the WireFactory class, but for creating gates rather than wires, see the code on the next page.

For example, the code below creates a disabler-circuit, (see Figure 13.9).

```
GateFactory gf;
Gate * cg = gf.MakeComposite();
Gate * ig = gf.MakeInverter();
Gate * ag = gf.MakeAndGate();
// connect wires, add gates, input and output wires, to cg
```

This class enables gates to be created using a factory, but it doesn't force client programs to use the factory. Nor does it stop clients from creating hundreds of factories. The second concern can be addressed using a design pattern called *Singleton*. A singleton class allows only one object to be created. Clients can have multiple pointers to the object, but there's only one object. The class `Singleton` in `singleton.h` illustrates how to do this.

Program 13.15 singleton.h

```
#ifndef _SINGLETON_H
#define _SINGLETON_H

// demo code for a singleton implementation

class Singleton
{
public:
    static Singleton * GetInstance();
    // methods here for Singleton behavior
private:
    static Singleton * ourSingleton;
    Singleton(); // constructor
};

Singleton * Singleton::ourSingleton = 0;

Singleton * Singleton::GetInstance()
{
    if (ourSingleton == 0)
    {
        ourSingleton = new Singleton(); // ok to construct
    }
    return ourSingleton;
}

Singleton::Singleton()
{
    // nothing to construct in this simple example
}

#endif
```

singleton.h

Show by example how client code uses a singleton object. Assume there's a void method `Singleton::DoIt()` and write code to call it. Explain how client programs are prevented from creating `Singleton` objects and how the class limits

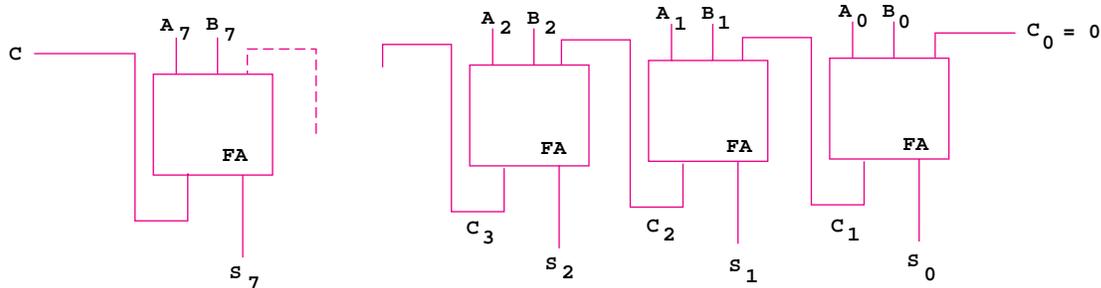


Figure 13.14 A ripple-carry adder for 8-bit numbers.

itself to creating one object. Then modify either your `GateFactory` or the existing `WireFactory` class to be singletons.

- 13.10** The circuit in Figure 13.14 is an 8-bit ripple-carry adder, a concrete version of the more general n -bit ripple-carry adder. The circuit adds two 8-bit numbers represented by A and B , where $A = A_7A_6A_5A_4A_3A_2A_1A_0$, and A_0 is the least-significant bit. The largest 8-bit value is 1111111 which is 255_{10} (base 10). Each box labeled FA is a full-adder, see Figure 13.13 for details. This ripple-adder is a 17-9-gate circuit, with 17 inputs: 8 bits for A , 8 bits for B , and the initial carry-in value, and 9 outputs: 8 bits for the sum and a final carry-out.

Write an English description for how the ripple-carry adder works. Note that the initial carry-in C_0 is set to zero. Other carries ripple through the circuit, hence the name. Then write a function `RippleAdder` to create and return a composite-gate representing an n -bit ripple-carry adder where n is a parameter to the function. Assume you have a function `FullAdder`. To test the function you'll need to implement the `FullAdder` function which in turn will require implementing a `HalfAdder` function.

- 13.11** In real circuits, electricity does not travel through a circuit instantaneously, but is delayed by the gates encountered. Different gates have different built-in delays, and the delays of the built-in gates affect circuits built up from these gates.

For example, we'll assume a delay of 3 time-units for an and-gate, 5 units for an or-gate, and 2 units for an inverter (you'll be able to change these values in the program you write). Assume a disabler-circuit as diagrammed in Figure 13.9 has the input to the and-gate from the outside on, the input to the inverter off, so that the output signal is on. If the inverter-input signal is set to true, the circuit's output will change to false five time-units later. There will be a 2-unit delay for the inverter followed by a 3-unit delay for the and-gate.

Develop a new class called `TimedGate` that acts like a gate, but delays acting for a set amount of time. This is a nontrivial design, so you'll need to think very carefully about how to incorporate delays into the circuit system. Assume you'll be using only `TimedGates`, not mixing them with regular gates. One way to start is shown in the following:.

```

class TimedGate : public Gate
{
public:
    // substitute me for g
    TimedGate(Gate * g, int delay);

    virtual int InCount() const
        {return myGate->InCount();}
    virtual int OutCount() const
        {return myGate->OutCount();}
    virtual Wire * InWire(int n) const
        {return myGate->InWire(n);}
    virtual Wire * OutWire(int n) const
        {return myGate->OutWire(n);}
    virtual string toString() const;
        // can use g's toString

    virtual Gate * clone()
    {
        return new TimedGate(myGate->clone(), myDelay);
    }
    virtual void Act(); // act with delay

protected:
    Gate * myGate;
    int myDelay;
};

```

This class can be used as-a gate. It forwards most requests directly to the gate it encapsulates as shown. The constructor and the `TimedGate::Act` function require careful thought.

A `TimedGate` object must remove the `Gate g` it encapsulates from the wires connected to `g`'s inputs. Then the `TimedGate` object substitutes itself for the the inputs. All this happens at construction.

In addition, you'll need to define some kind of structure that stores timed events so that they happen in the correct order. In my program I used a static `EventSimulator` object that all `TimedGates` can access. Events are put into the simulator, and arranged to occur in the proper order. Again, you'll need to think very carefully about how to do this.

13.12 The circuit in Figure 13.15 is designed to control an elevator. It's a simple circuit designed to direct the elevator up or down, which are the circuit's outputs. The inputs are the current floor and the requested floor. The diagram shows a circuit for an elevator in a four-story building. The current floor is specified by the binary number C_1C_0 , so that 00 is the first floor,¹² 01 is the second floor, 10 is the third floor, and 11 is the fourth floor.

¹²This is a book on C++, so floors are numbered beginning with zero.

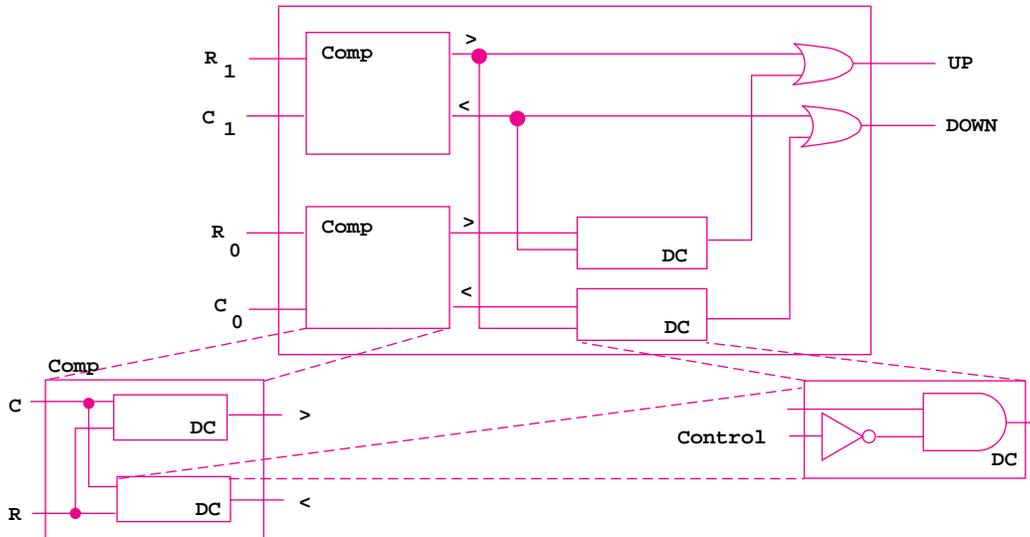


Figure 13.15 A circuit for choosing which direction an elevator travels. The inputs labeled C are the current floor (2 bits) and the R inputs are for the requesting floor.

The digit C_1 is the most significant digit. Similarly, $R_1 R_0$ is a binary representation of the requested floor.

The purpose of the circuit is to direct the elevator up when the requested floor is greater than the current floor, and down when the requested floor is less than the current floor. Write an English description of why the circuit works. Be attentive to the order of inputs to the comparator gates and see Fig 13.10 and Figure 13.9 and the associated descriptions.

Write a truth table by hand for a 4-input 2-output circuit, or build the circuit with a program and have `GateTester::Test` print the truth table for the circuit. Try to generalize the circuit to a building with 2^N floors rather than four floors.