Design Patterns for Data Structures

Dung X. Nguyen J. Stanley Warford

August 30, 2021

Contents

Preface

The goal of *Design Patterns for Data Structures* is threefold:

- To teach the data structures central to computer science
- To teach object-oriented design patterns
- To teach C++ programming

This book is unique because it combines the first two goals into one unified approach in a novel way.

Object-oriented design patterns are usually applied to programming in the large. Consequently, teaching OO design patterns is traditionally postponed until later in the curriculum, when they can be applied to large software projects in software engineering or capstone courses. One problem with this approach is that students learn OO design patterns late in their academic careers and so have less time to assimilate them. Another problem is that the sheer amount of code in such large projects can obscure the salient features of the OO design patterns.

This book applies OO design patterns to programming in the small. That is, the classic data structures are presented in their traditional implementations, but then are re-implemented using the patterns of the "Gang of Four".[1] Specifically, this text teaches the following OO design patterns:

- The Adaptor pattern
- The Decorator pattern
- The Iterator pattern
- The Template Method pattern
- The Composite pattern
- The State pattern
- The Visitor pattern
- The Factory pattern

Hence, students learn OO design patterns early in their academic careers, and the salient features of the patterns are not obscured by large amounts of code.

Another advantage of this approach of combining OO design patterns with data structures is its efficiency in curricula with perennial constraints on the number of academic units permitted for a major course of study. In most computer science departments, educators do not have the luxury of offering any number of courses they feel

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

would be desirable for their computer science majors. *Design Patterns for Data Structures* kills two birds with one stone. In a single course, students learn all three subjects—data structures, object-oriented design patterns, and C++.

Historically, OO design patterns evolved from large-scale software projects. Because these patterns are not usually applied to data structures, it is reasonable to question the approach. Here are some possible arguments against the approach used in *Design Patterns for Data Structures*.

"Applying OO design patterns to data structures is overkill, and it makes things more complicated." It is true that implementations of the classic data structures using OO design patterns take more lines of code, and by that measure are more complicated. The flip side to that disadvantage is that *conceptually* OO design patterns can actually make the implementations *simpler*. Our students often remark how they prefer implementing, for example, a binary tree using the composite state design pattern as opposed to the traditional pointer-centric implementation, because they find it easy to conceptualize the solutions.

"Applying OO design patterns to data structures is not used in the real world." This is also a valid criticism, because many implementations of the data structures in this text are definitely *not* used in the real world. We accept this criticism as a tradeoff and view its cost to be acceptable to achieve the above benefits. The book emphasizes this point throughout, reminding the reader of its goals, so that students will have an accurate impression of how traditional data structures are implemented in the real world. It generally presents the classic implementation first, followed by the OO design pattern implementation.

"Applying OO design patterns to data structures does not use the Standard Template Library." The purpose of a data structures course is to learn the implementation of the classic data structures and the algorithms that manipulate them. Learning to use a given library does not further that goal. The book comes with its own software library and problem sets that are designed to teach data structures and their algorithms.

At this point in our students' educational careers, we cannot expect them to design elegant class structures. The educational philosophy here is for them to experience good class designs, which they can then emulate later. To that end, the book uses an extensive suite of software known as the dp4ds Distribution, which is a requirement for the effective use of the book. This software distribution has the following features:

- Code walkthroughs in the text are descriptions of code fragments in the Distribution. Hence, students are assumed to have access to the complete code from the Distribution. The exercises from the text are assignments to fill in missing implementations in the software Distribution.
- The Distribution has an extensive set of unit tests for every software assignment. The drivers for these tests are command-line oriented. Students can run the drivers in interactive mode to experiment with the data structures, or they can redirect the input to come from the unit tests.
- The `shared_ptr` type is used for pointers with automatic garbage collection throughout the entire Distribution.
- The Distribution is designed for standard C++17 and is platform-independent. It is compatible for C++ development with common IDEs such as Eclipse, NetBeans, and CLion (our current recommendation). Our students typically use Mac, Windows, and Linux systems to complete their assignments.

The concepts of a data structure and of an object-oriented design pattern to implement it are both based on abstraction. A data structure is the implementation of an abstraction that provides the user with a set of operations to assist in storing and retrieving data. A design pattern is a useful programming technique that aids the implementation. The first chapter emphasized the centrality of abstraction.

There is an extensive appendix that contains a review of C++. It covers all the necessary procedural parts of the language — syntax, primitive types, program structure — to enable a student with programming experience in some other language such as Java to begin with Chapter 1.

Chapter 1

Abstraction

Abstraction is based on the concept of layers in which the details of one layer of abstraction are hidden from layers at a higher level. A computer scientist uses abstraction as a thinking tool to understand a system, to model a problem, and to master complexity.

The concept of abstraction is pervasive throughout computer science, and is especially important in software design. Object orientation is one of the more recent software technologies to harness the power of abstraction. This chapter introduces the abstraction process, on which the design principles of the rest of the book are based. The deep significance of the concept of abstraction can hardly be overestimated, so this beginning chapter is essential to the remainder of the book. Later chapters apply the design principles introduced here to the problem of data structure specification and implementation.

1.1 Objects and Classes

Abstraction is a process. This section describes the process using the tool of object orientation with the C++ language. The history of computer science shows a steady progression from lower levels of abstraction to higher levels. When the electronic computer was first invented in the mid twentieth century, there was no assembly language much less the higher level languages with which we are familiar today. It is no accident that the historic evolution is toward progressively higher levels of abstraction instead of the other way around. Human intellectual progress shows that generalities are usually discovered from many specific observations. It is only with hindsight that you can start with the general case and deduce specific consequences from it.

Data abstraction

Plato, in his theory of forms, claimed that reality ultimately lies in the abstract form that represents the essence of individual objects we sense in the world. In The Republic, written in the form of a dialogue between Socrates and a student, he writes:

> Well then, shall we begin the enquiry in our usual manner: Whenever a number of individuals have a common name, we assume them to have also a corresponding idea or form: do you understand me?
>
> I do.

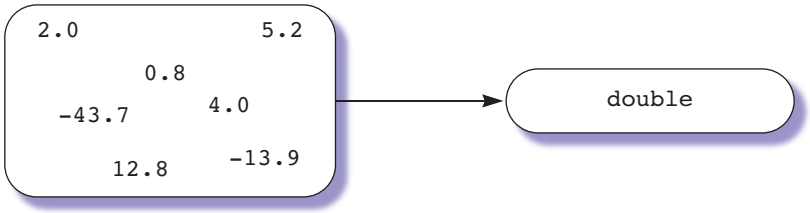


Figure 1.1 Type abstraction for type `double`. In the C++ programming language, an expression of type `double` must have as its value one of many possible specific values that define the type. The same principle holds for other types. For example, the values `true` and `false` define type `bool`.

> Let us take any common instance; there are beds and tables in the world—plenty of them, are there not?
>
> Yes.
>
> But there are only two ideas or forms of them—one the idea of a bed, the other of a table.
>
> True.
>
> And the maker of either of them makes a bed or he makes a table for our use, in accordance with the idea—that is our way of speaking in this and similar instances—but no artificer makes the ideas themselves: how could he?
>
> Impossible.

Plato's consideration between the specific and the general exemplifies the abstraction process. Another example of the abstraction process is the concept of type in programming languages. Consider all the possible real values, such as 2.0, −43.7, 5.2, 0.8, and so on. In the same way that Plato considered many different instances of a table to be representations of a single abstract table, from a computation point of view the collection of all possible real values defines a single abstract type double. Figure 1.1 shows the abstraction process, known as *type abstraction*, for type `double`. A type is defined by a collection of values. Each value, such as 5.2 in the box on the left, is specific, while the type `double` is general.

In the history of computing languages, types emerged as one of the first steps toward higher levels of abstraction. At the machine level, which must be programmed with machine language or its equivalent assembly language, there are no types other than the bit patterns of pure binary. With assembly language, you have unlimited freedom to interpret a bit pattern any way you choose. The same bit pattern in a specific memory location can be interpreted as an integer and processed with the addition circuitry of the processor. It can be interpreted as a character and sent to a Web page as such. It can even be interpreted as an instruction and executed.

In C++, every variable has a name, a type, and a value. The name is an identifier, defined by the syntax rules of the language. The type is supplied by the language. Both the name and the type of a variable are determined when the software designer writes and compiles the program. The value of a variable, on the other hand, is stored in the main memory of the computer as the program is executing. The value stored is one of the values that defines the type.

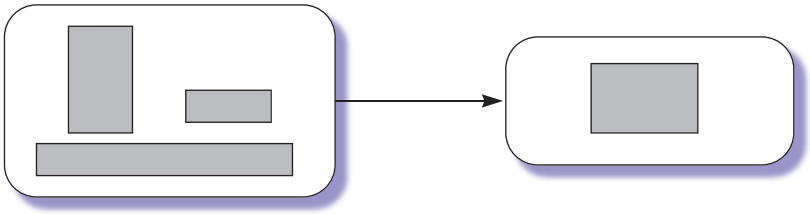


Figure 1.2 Structure abstraction to abstract from specific shapes of many different sizes to a single shape with a general size.

The compiler enforces type compatibility, which is a restriction on the freedom of programmers that they do not have with assembly language. The abstraction process frequently imposes a loss of freedom because the nature of abstraction is the hiding of detail. Programmers then have no access to the details that are hidden. With the advent of types to restrict the value that a variable can have to some mathematical entity like a real number comes the inability to consider the bit pattern behind the value. But the restriction of freedom to access low-level details is also liberation from the necessity to do so. Abstraction is powerful because the limitation it places on the programmer's ability to access low level details at the same time frees the programmer from that requirement.

The abstraction process permits the grouping together of specific real values into a type because each value shares certain characteristics with all the other values. For example, each value has a sign and a magnitude. Any value can be combined with any other value with the arithmetic operators like multiplication. And any value can be compared with any other value to determine whether the first is less than, equal to, or greater than the second. If it were not for these common properties among individual values, the grouping together of them to define a type would not be useful.

Furthermore, the collection of many specific numeric values to make a general type is useful in a programming language because it models the same process in the real world. For example, the type double in C++ corresponds to the notion of a real number in mathematics. All computer applications exist to solve problems in the human world. The first step toward solving any problem is to model it with the machine. There are usually approximations to the model, which may make the solution approximate. For example, there are only a finite number of real values that a computer can store while there are an infinite number of real values in mathematics. Nevertheless, one source of power of the abstraction process in computing is that it can mirror the same process in the human world and so serve as a model to compute the desired solution.

The next step toward higher levels of abstraction in programming languages occurred when languages gave programmers the ability to create new types as combinations of primitive types. Collections of primitive types are known as records or structures in most programming languages. The corresponding abstraction process is called *structure abstraction*.

For example, Figure 1.2 shows geometrically how the collection of all possible rectangles define a single rectangle type. The abstraction process parallels the process of defining a type as an aggregate of values. An individual rectangle is characterized by its length, say 2.0, and width, say 5.2. This is not the only possible rectangle. Mathematically there are an infinite number of rectangles, each with its own length and width.

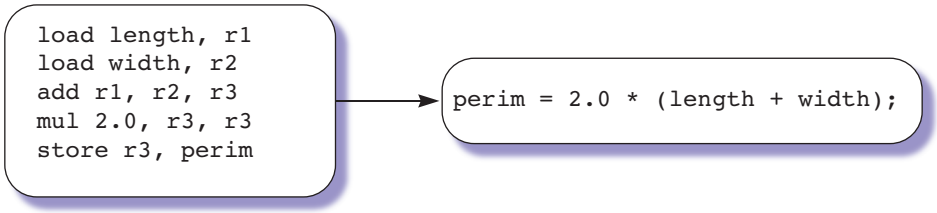


Figure 1.3 Statement abstraction for the assignment statement. In C++, the assignment operator evaluates the expression on its right hand side and gives the value to the variable on the left hand side.

Because computers can only store a finite number of real values in a memory cell, the number of possible rectangles that can be characterized in the machine is finite.

Programmer-defined types are powerful because they allow the programmer to conveniently model the problem to mirror the situation in the problem domain. For example, an airline reservation system might need to store a collection of information for each ticket it sells, say the passenger's name, address, flight date, flight number, and price of the ticket. Collecting all these types into a single programmer-defined type allows the program to process a ticket variable as a single entity.

Each of the rectangles in Figure 1.2 is specified by its length and width. In C++, you could define a new type `Rectangle` as a structure that contains two real numbers for storing those dimensions.

```
struct Rectangle {
    double length;
    double width;
};
```

You could declare an individual rectangle as a variable of type `Rectangle`.

```
Rectangle myRectangle;
```

To set the length of `myRectangle` to 2.0 use a period to separate the variable name from the `struct` field name as follows.

```
myRectangle.length = 2.0;
```

Computation abstraction

Abstraction of data is only one side of a two-sided coin. The other side is abstraction of computation. At the lowest level between programming languages and the machine is *statement abstraction*.

All computers consist of a central processing unit (CPU) that has a set of instructions wired into it. The instruction set varies from one computer chip maker to another, but all commercial CPUs have similar instructions. CPUs contain cells called registers that store values and perform operations on them. The collection of the operations specifies a computation.

Typical instructions are `load`, `add`, `mul`, and `store`. The `load` instruction gets a value from main memory and stores it in a register of the CPU. The `add` instruction adds


```
double perimeter(Rectangle r) {
    return 2.0 * (r.length + r.width);
}
```

→

```
cout << perimeter(myRectangle);
```

Figure 1.4 Procedure abstraction for the computation of the perimeter of a rectangle. The box on the left contains the C++ code for defining a function named `perimeter`. The box on the right contains an example of how the function is called.

the content of two registers. The `mul` instruction multiplies the content of two registers. The `store` instruction puts a value from a register of the CPU into main memory.

Before the advent of high-level languages, programmers wrote their programs using the individual instructions of the instruction set of the particular CPU on which the program was designed to run. Figure 1.3 shows an example of a sequence of instructions for some hypothetical CPU that computes the perimeter of a rectangle. The first two instructions load the value of length into register r1 and the value of width into register r2. The next instruction adds the content of r1 to r2 and puts the sum in register r3. Then, 2.0 is multiplied by the content of r3 with the result placed back in r3, after which it is stored in main memory in the location reserved for variable `perim`.

The language illustrated by this sequence of instructions is called assembly language. When you program in assembly language you must consider the details of the CPU—how many registers it has, how to access them, and which values you want to store in which registers. In a high-level language, however, all those details are hidden. The compiler abstracts them away from the view of the programmer, so that the programmer need only write the single assignment statement

```
perim = 2.0 * (length + width);
```

With statement abstraction, even the structure of the CPU is hidden. The programmer does not need to know about registers or hardware instruction sets. A single assignment statement in C++ is translated by the compiler to several instructions in assembly language. One statement in a high-level language is defined by many statements at the machine level like one type in a high-level language is defined by many possible values at the machine level.

Corresponding to structure abstraction on the data side of the coin is *procedure abstraction* on the computation side. In the same way that high-level languages allow you to collect variables into structures to create a new data type, they allow you collect statements into procedures to create a new computation. The corresponding abstraction process is procedure abstraction.

Figure 1.4 shows procedure abstraction for the computation of the perimeter of a rectangle. The C++ computation of the perimeter of an arbitrary rectangle is encapsulated in a function with formal parameter `r` whose type is `Rectangle`. Any time the programmer needs to compute the perimeter, for example to stream it to `cout`, a simple call to the function is all that is required. The computation need only be done once, freeing the programmer from having to remember those details whenever the computation is required.

For example, if you have two variables—`myRectangle` and `yourRectangle`—both of type `Rectangle`, you can output their perimeters with

```
cout << perimeter(myRectangle);
```

and

```
cout << perimeter(yourRectangle);
```

The details of the computation are hidden in the function calls.

The benefit of procedure abstraction can be even more apparent when the procedure contains many statements. For example, function `gcd()`

```
int gcd(int m, int n) {
    if (0 == n) {
        return m;
    } else {
        return gcd(n, m % n);
    }
}
```

computes the greatest common divisor of two integers. The algorithm returns the first integer if the second integer is 0. Otherwise, it recursively returns the greatest common divisor of the first integer and the second integer modulo the first. If you need to compute the greatest common divisor of two integers, say `num` and `denom` that represent the numerator and denominator of a fraction, you could write the assignment

```
temp = gcd(num, denom);
```

As with statement abstraction in Figure 1.3, this one statement at a high level causes the execution of many statements at a lower level.

Class abstraction

The next step in the evolution of programming languages toward higher levels of abstraction was the combination of data abstraction with computation abstraction to produce *class abstraction*. Consider again the rectangles in Figure 1.2 and imagine what sort of processing might be required for such geometric figures. A rectangle might represent part of a building like the interior wall of a room or a door. If the walls and doors are to be painted your program would need to compute the area of each rectangle to determine the amount of paint required. Or a rectangle might represent a piece of land around which a fence is to be erected. Your program would then need to compute the perimeter to determine the amount of material required for the fence.

Before the advent of object-oriented programming, the function to compute the area or the perimeter of a rectangle would exist separately from its dimensions. For example, you might have this function to compute the perimeter.

```
double perimeter(Rectangle r) {
    return 2.0 * (r.length + r.width);
}
```

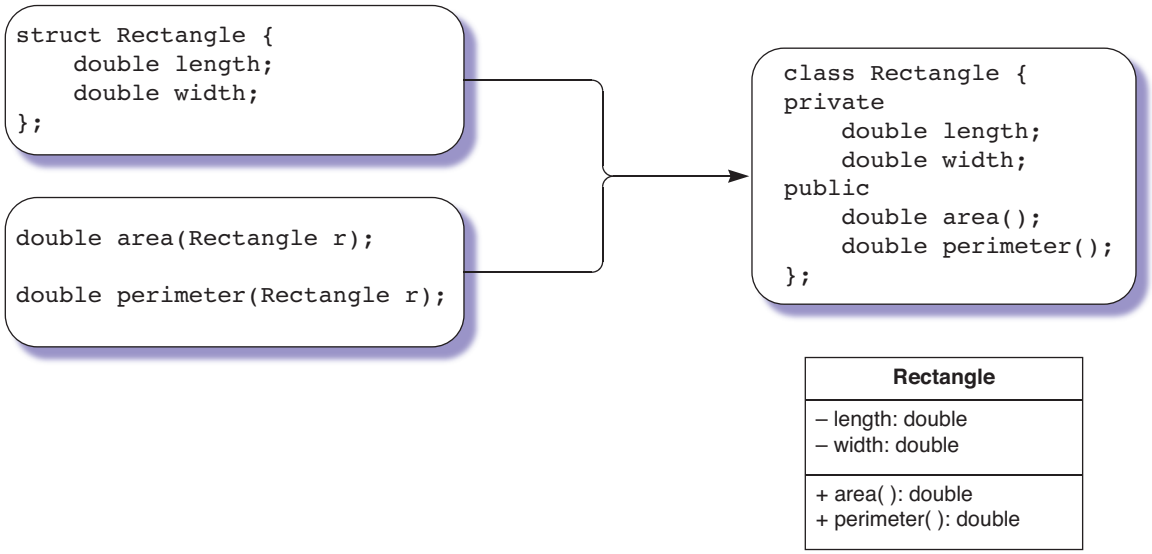



Figure 1.5 Class abstraction that combines the structure abstraction of Figure 1.2 with the procedure abstraction of Figure 1.4. The box in the lower part of the figure shows a Unified Modeling Language (UML) depiction of the class.

The rectangle would be passed as a parameter to the function, and then its constituent parts—its dimensions—would be used to compute the perimeter. For example, to output the perimeter of variable `myRectangle` you would write

```
cout << perimeter(myRectangle);
```

where `myRectangle` is the actual parameter corresponding to formal parameter `r`.

Object orientation includes in the abstraction process not only the aggregation of data, but the aggregation of computation as well. It is a viewpoint that shifts the focus from an external operation that requires the input of data about the rectangle, to an internal operation that is part of the rectangle itself. This is a significant shift in focus. Computing the perimeter is no longer something that you do to a rectangle. It is something the rectangle does for you. The rectangle knows its dimensions and should be the party responsible for computing its perimeter.

In Figure 1.2, each individual rectangle on the left has an area and a perimeter in addition to its length and width. The area and perimeter are not data values that are independent from the dimensions. So, their values should not be stored the same way the dimensions are stored, but they should be computed from the dimensions. In object-oriented design, the functions to compute the area and perimeter are no longer external to the type, but are internal. They literally become part of the type. Figure 1.5 shows how class abstraction merges the dimensions and the functions into class `Rectangle`.

To emphasize the shift in focus when a function is bound to a type, object-oriented designers established a new set of terminology. Roughly speaking, in object-oriented terminology

UML terminology	C++ terminology
class	class
superclass	base class
subclass	derived class
attribute	data member
operation / method	member function
visibility	access specifier
parameterized class	template
abstract	pure virtual

Figure 1.6 Object-oriented terminology for UML and C++.

- *class* corresponds to *type*
- *object* corresponds to *variable*
- *method* corresponds to *procedure* or *function*

That is, an object has a class, like a variable has a type. It is more usual to state that an object is an instantiation of a class rather than to state that an object has a class.

Unified Modeling Language

Object-oriented design is widespread in the computing industry and is not confined to any one language such as C++. In the late 1980s and early 1990s, many object-oriented analysis and design methods emerged to aid the software development process. Different people devised different methods, but because they all attempted to solve similar problems they shared common characteristics.

For a while the object-oriented design community was split among several warring factions who could not agree on a common standard for communicating object-oriented concepts. Eventually, the major players in the debate teamed up and merged their methods into what has become an industry standard called the Unified Modeling Language (UML). One part of UML is a graphic language called a class diagram that is independent of any specific programming language.

Part of the UML effort was to establish a common vocabulary for communicating object-oriented concepts. Unfortunately, each programming language has its own terminology that in many cases predates the UML effort and that differs from the UML vocabulary. Figure 1.6 lists some UML terms and the corresponding terms in C++.

Figure 1.5 includes a UML depiction of the rectangle class declared in the same figure. A box in a UML class diagram represents a class. Generally, a class box contains three compartments—the class name, the class attributes, and the class operations. The top compartment contains the name of the class in a bold typeface.

The middle compartment contains the attributes of the class. Attributes in UML terminology correspond to the data part of a class. In Figure 1.5, the attributes of class

Rectangle are

– length: double
– width: double

On each line, the visibility marker comes first, followed by the name of the field, followed by a colon, followed by the field's type. In the above examples, the dash character indicates that the fields are private. Private attributes are ones that are not accessible by any other functions except those that are bound to the class. The protection provided by private attributes is described in more detail later in this chapter.

The bottom compartment contains the operations of the class. Operations in UML terminology correspond the methods of the class, known as member functions in C++. In Figure 1.5, the operations of class Rectangle are

```
+ area( ): double
+ perimeter( ): double
```

On each line, the visibility marker comes first, followed by the name of the operation, followed by its formal parameter list enclosed in parentheses, followed by a colon, followed by the type returned by the operation. If the operation is a procedure, corresponding to a C++ function that returns void, the returned type and colon are simply omitted. In the above examples, the plus symbol indicates that the operations are public. Public operations are ones that can be called by any other function, such as a main program.

C++

The syntax of C++ for binding methods to data to make a class is similar to the syntax for a `struct`. The similarity is not coincidental, because the `struct` is what allows for the grouping of data in the abstraction process. It is simply extended to allow methods to be included in the grouping as well as data.

Compare the C++ declaration of structure Rectangle from page 4 that abstracts only the data

```
struct Rectangle {
    double length;
    double width;
};
```

with the corresponding declaration in Figure 1.5 that abstracts the operations as well as the data.

```
class Rectangle {
private:
    double length;
    double width;
public:
    double area();
    double perimeter();
};
```


In place of the keyword `struct` is the keyword `class`. In addition to grouping the `length` and `width` data, you group methods by including the function prototype for each method within the braces of the class. You indicate the access privileges of an item in a class with the `private` and `public` reserved words. Each word is followed by a colon.

Declaring an object is analogous to declaring a variable. Compare this declaration of object `myRectangle`

```
Rectangle myRectangle;
```

with that of variable `myRectangle` on page 4. The declarations are the same.

The syntax for implementing and calling a method, however, differs slightly from that for implementing and calling a function. When you define a method, you must first specify the class to which it is bound. For example, the definition of the `perimeter` method is

```
double Rectangle::perimeter() {
    return 2.0 * (length + width);
}
```

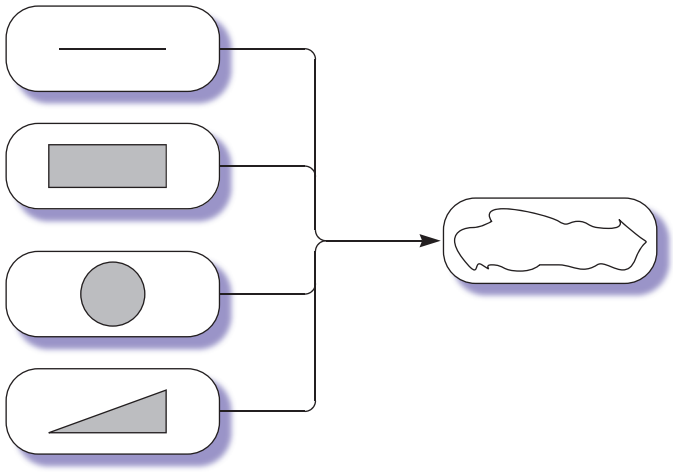
Compare this definition with the definition of function `perimeter()` on page 7. When you define a method, you must include the name of the class to which the method is bound just before the name of the method and separated with the double-colon scope operator `::`. The parameter list for this method is empty. How, you might well ask, does the method know which rectangle to get the height from? The function on page 7 uses `r.length`, where `r` is passed as a parameter. This method simply uses `length`. Whose length is it?

The answer is that there is, in effect, an implicit formal parameter not shown in the definition of the method. The corresponding actual parameter is, however, explicit when the method is called. To output the perimeter of object `myRectangle`, you write

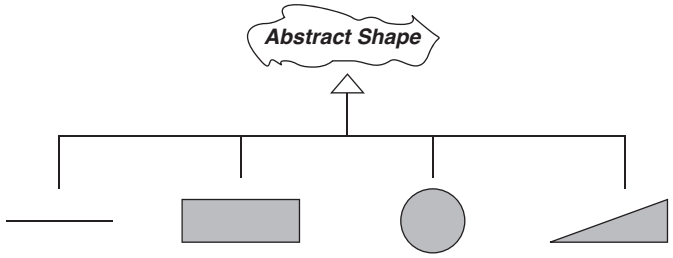
```
cout << myRectangle.perimeter();
```

Compare this method call with the function call on page 7. There, `myRectangle` is an actual parameter enclosed in parentheses. Here, it is also an actual parameter, but it is not enclosed in parentheses. It is placed in front of the method name and is separated from it by a period. This notation is consistent with the fact that the method is part of the class alongside the data, as it is accessed with the same period syntax. In the definition of the method, the expression `length + width` refers to the length and width of the actual parameter `myRectangle` whose formal parameter does not appear in the definition.

The difference in syntax for defining and calling a method compared to a function does not illustrate the power of object oriented design. After all, there is no inherent benefit to putting an actual parameter in front of a method name instead of enclosing it in parentheses after a function name. The only thing the object-oriented syntax does is to emphasize that functions are bound to classes along with the data. The real power of object-orientation comes with yet another level of abstraction—behavior abstraction with polymorphism.



(a) Behavior abstraction for shapes rendered geometrically.



(b) Behavior abstraction for shapes rendered with the UML symbol for inheritance.

Figure 1.7 Using inheritance to abstract from specific shapes of many different types to a single shape with a general type.

1.2 Abstract Classes and Inheritance

This section presents the next step to higher levels of abstraction. It introduces a complete C++ program that illustrates the highest level of abstraction in object-oriented design.

An abstract class

The abstraction process consists of collecting together many items that share a common characteristic and creating a new item that is a general representation of each specific item. You can collect many individual real values such as 2.0, 5.2, and 12.8 to create an abstract type `double`. A variable of type `double` has one of the collection of values. You can collect two doubles—one each for length and width—and put them together with methods to create a rectangle class. A specific rectangle will have values for the two doubles and will have methods for computing its area and perimeter.

But there are shapes in the universe other than rectangles. There are circles, lines,

right triangles, and many others. What do these shapes have in common? They are certainly not all specified by length and width as is the rectangle. A circle, for example, is specified by its radius. Suppose you want to take a further step towards abstraction and collect several different shapes together to form an abstract shape. What is common that can be abstracted out?

Because dimensions for different objects are specified differently, you cannot include the dimensions in the abstract shape. However, all closed shapes have an area and a perimeter. So, you can at least include those. You must be careful, however, because the algorithm for computing the area of a circle is not the same as the algorithm for computing the area of a right triangle. Even though the abstract shape will specify a method for computing the area, the method cannot implement it because the algorithm depends on the specific object.

Figure 1.7(a) is a geometric representation of the abstraction process. Many different shapes are collected to form an abstract shape represented by the cloud in the box on the right. Figure 1.7(b) is a representation for the same abstraction process, but in a graphic form more closely resembling a UML class diagram. The symbol △ is the UML notation for inheritance, which is the relationship between a specific shape and the general shape. Each specific shape, such as the rectangle, is a subclass of the abstract shape class, which is called the superclass. A subclass inherits from its superclass. In C++ terminology, the superclass is known as the base class and the subclass is known as the derived class.

Figure 1.8 shows how to declare an abstract class in C++. It adds a few more methods to our geometric shape example, `scale()` and `display()`, and the virtual destructor `~aShape`.

The name of the class in the C++ code is `AShape`. The UML standard notation for an abstract class is to render the name of an abstract class in bold slanted type and the name of its methods in slanted type. The C++ syntax for a formal parameter is to have the type followed by the name separated by a space. The UML syntax is to have the name followed by the type separated by a colon. The C++ syntax for the returned type is to have the type precede the method name separated by a space. The UML syntax is to have the returned type follow the method name separated by a colon.

In C++, the first line after the opening brace

```
public:
```

states that the following items are public. That is, the following items are available or accessible for client programs to use.

The first item in the public list

```
virtual double area() = 0;
// Post: The area of this shape is returned.
```

declares a method named `area()`. The key word `virtual` at the beginning of the declaration makes it possible to invoke the method with polymorphism, a concept that will be illustrated in more detail later in the next section. The notation `=0` at the end is a rather curious syntactic rule of C++, because it looks somehow like zero is being assigned to `area()`. But nothing of the kind is implied by that notation. Instead, the notation indicates that a programmer must override `area()` when producing the corresponding concrete method. The task of writing such code is left to the programmer of the specific line, rectangle, circle, and right triangle derived classes.


```
#ifndef AShape_hpp
#define AShape_hpp

#include <iostream> // ostream.
using namespace std;

class AShape {
public:
    virtual ~AShape() = default;
    // Virtual destructor necessary for subclassing.

    virtual double area() = 0;
    // Post: The area of this shape is returned.

    virtual double perimeter() = 0;
    // Post: The perimeter of this shape is returned.

    virtual void scale(double factor) = 0;
    // Pre: factor > 0.0
    // Post: This shape's dimensions are multiplied by factor.

    virtual void display() = 0;
    // Post: This shape's name and dimensions are printed to cout.

    virtual void promptAndSetDimensions() = 0;
    // Post: This shape's dimensions are prompted and set.
    // No dimension is negative.
};

#endif
```

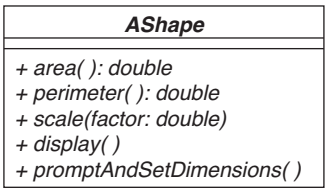


Figure 1.8 `AShape.hpp`. Declaration of the abstract shape class of Figure 1.7. The box is the UML class diagram corresponding to the C++ code.

The keyword `virtual` together with the `=0` notation make method `area()` what is known in C++ as a *pure virtual* function. The idea is that the information in this declaration specifies what the method should do, and not how it should do it, a task that will be different for each derived class. In this example, every derived class of `AShape` must have a method named `area()` that returns a double precision real value and has no parameters in its parameter list.

The comment line specifies the postcondition and serves as documentation of what the method should do. In this example, the purpose of the method is to compute the area of the shape. You can see that it would be impossible to write the code for the general case because the computation for the area of a shape depends on a specific shape. For a circle, the area is pi times the square of the radius, while for a right triangle it is half the base times the height.

Occasionally, a method will have a precondition as well as a postcondition as does the `scale()` method. A precondition is a statement that must be true for the method to produce correct results. A precondition in the documentation of a program corresponds to a precondition of a Hoare triple. For example, the precondition for a method that computes the real square root of a number is that the number be nonnegative, because you cannot take the square root of a negative number. The methods in this book all have an implied precondition that the object exists, because you cannot compute with a nonexistent object. The specification for area could have been written

```
virtual double area() = 0;
// Pre: This shape exists.
// Post: The area of this shape is returned.
```

To save space the existence precondition will always be omitted, but implicit. If a client program invokes a method without satisfying the precondition, the program aborts. It is the responsibility of the client to insure that the precondition is met when the method is called.

The fundamental notion of abstraction is hidden detail. Class `AShape` is abstract because it hides the details of the computations that must be done by the methods for the specific shapes. The interface states that all specific shapes must have at least the four methods: `area()`, `perimeter()`, `scale()`, and `display()`. These methods represent behavior abstraction in the abstraction process. The details of how the computations are done are hidden at this level of abstraction.

A concrete class specification

Figure 1.9 shows the content of the header file for one of the concrete classes, the `Rectangle`. The include statement

```
#include "AShape.hpp"
```

includes the header file for `AShape` shown in Figure 1.8. Hence, before the compiler proceeds with the code in this header file, it has scanned the declaration for the `AShape` class.

The declaration for `Rectangle`

```
class Rectangle : public AShape
```

has a colon between the name of the class that is being declared as the derived class and the name of the base class from which it is derived. The colon in C++ corresponds to the UML symbol for inheritance. The keyword `public` before the base class indicates public inheritance. Public inheritance means that all the methods that are public in the base class will also be public in the derived class. Although C++ has two other forms of inheritance, protected and private, this book has no occasion to use either.

The shape of a rectangle is specified by its length and width. Corresponding to these two dimensions are the two attributes in the private part of the Rectangle class

```
private:
    double _length;
    double _width;
```



```
#ifndef Rectangle_hpp
#define Rectangle_hpp

#include "AShape.hpp"

class Rectangle : public AShape {
private:
    double _length;
    double _width;

public:
    explicit Rectangle(double length = 0.0, double width = 0.0);
    // Pre: length >= 0.0 and width >= 0.0.
    // Post: This rectangle is initialized with
    // length length and width width.

    double area() override;
    double perimeter() override;
    void scale(double factor) override;
    void display() override;
    void promptAndSetDimensions() override;
};

#endif
```

Figure 1.9 `Rectangle.hpp`. Specification of the concrete `Rectangle` class. Our C++ style convention is to always begin the name of an attribute with the underscore character as with `_length` in the listing.

The variable `_length` stores the length of the rectangle and `_width` stores its width. Items that are in the private part of a class specification are not directly accessible to other programs, although they are indirectly accessible through the operations that are provided in the public part.

C++ provides private and public access to help manage the development of software when more than one programmer is on the development team. It is common for a class to be written by one programmer and used by another. The user of the class may not be familiar with the details of the private part. To allow the user access to the private part is to risk the possibility that he may modify it somehow erroneously. For example, this object stores its length and width and insures that these values are always nonnegative. If the client had public access to the values it could change one of them to a negative value, after which computation of the area would produce meaningless results. The purpose of the private part is for protection of the object against unauthorized, possibly erroneous, manipulation.

Figure 1.9 shows the public part of the `Rectangle` class, consisting of the methods that are specified in the base class plus one other called a constructor.

```
Rectangle(double length = 0.0, double width = 0.0);
```

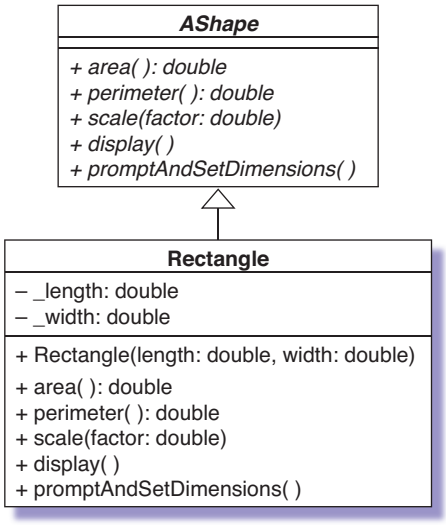



Figure 1.10 The UML diagram for the `Rectangle` class of Figure 1.9. The UML syntax for a private item is to precede it with the – symbol, and for a public item is to precede it with the + symbol.

```
// Pre: length >= 0.0 and width >= 0.0.
// Post: This rectangle is initialized with
// length length and width width.
```

A constructor is a method that has the same name as the class, and that has no specified return type, not even void. In this example, the name of the class is `Rectangle`, and the name of the constructor is also `Rectangle()`.

A class can have more than one constructor as long as the parameter lists of the different constructors are different. Unlike other methods, constructors are not called explicitly. Instead, C++ forces them to be called implicitly before the object is first used. The C++ compiler inserts a call to the constructor when an object is declared, or when an object is created with the `new` or `make_shared<>()` function. The purpose of a constructor is to give an object initial values when it is created.

This constructor has default parameters indicated by `= 0.0` in the formal parameter list. If the actual parameter list is empty, the constructor is called with `length` and `width` both having their default values of 0.0. If the actual parameter list has one value, say 5.7, then the constructor is called with `length` having value 5.7 and `width` having its default value of 0.0. (Default parameters are described in more detail on page 488 in the appendix.)

The `override` specifier in

```
double area() override;
```

indicates to the compiler that this method should be declared in the base class `AShape`. If it is not, the compiler will issue an appropriate error message.

It is common for a derived class to have methods in addition to the ones that it inherits from its base class. The documentation convention in this book is to specify the


```
#include "Rectangle.hpp"
#include "Utilities.hpp"

Rectangle::Rectangle(double length, double width) {
    if (length < 0.0 || width < 0.0) {
        cerr << "Rectangle precondition violated: "
             << "length and width cannot be negative." << endl;
        throw -1;
    }
    _length = length;
    _width = width;
}

double Rectangle::area() {
    return _length * _width;
}

double Rectangle::perimeter() {
    return 2.0 *(_length + _width);
}

void Rectangle::scale(double factor) {
    cerr << "scale: Exercise for the student." << endl;
    throw -1;
}

void Rectangle::display(ostream &os) {
    os << "Rectangle" << endl;
    os << "Length: " << _length << endl;
    os << "Width: " << _width << endl;
}

void Rectangle::promptAndSetDimensions() {
    _length = promptDoubleGE("Length?", 0.0);
    _width = promptDoubleGE("Width?", 0.0);
}
```

Figure 1.11 `Rectangle.cpp`. Implementation of the `Rectangle` class. The code for method `scale()` in this and all the other shapes is left as an exercise for the student.

preconditions and postconditions for the methods in the abstract class only once, and not repeat them in the header files of any of the derived classes. Methods that are new to the derived class, such as the Rectangle constructor, are documented in the header file of the derived class.

Figure 1.10 shows the UML diagram that corresponds to the `Rectangle` class of

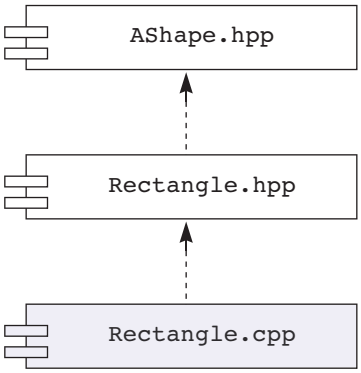


Figure 1.12 Component diagram showing the dependencies between the files in Figures 1.8, 1.9, and 1.11.

Figure 1.9. The △ symbol shows that class `Rectangle` inherits from class `AShape`. The name of the class `Rectangle` is in the top compartment of the box, the attributes `_length` and `_width` are in the middle compartment of the box, and the operations are in the bottom part of the box.

A concrete class implementation

Figure 1.11 shows the implementation of the `Rectangle` class contained in file `Rectangle.cpp`. The include statement

```
#include "Rectangle.hpp"
```

instructs the compiler to scan the file `Rectangle.hpp` before scanning the code in this file. However, Figure 1.9 shows that file instructing the compiler to scan the file `AShape.hpp` before scanning its code.

In UML terminology, a physical unit of code, such as the content of source code in a file, is a *component*. A component diagram shows the dependencies between components with dashed arrows. Figure 1.12 is a component diagram that shows the compile time dependencies between the files in Figures 1.8, 1.9, and 1.11. The arrow from `Rectangle.cpp` to `Rectangle.hpp` means that `Rectangle.cpp` depends on `Rectangle.hpp` by virtue of the include statement in Figure 1.11.

The first line of method `area`,

```
double Rectangle::area()
```

begins with the type of the value returned, `double` in this case. After the value returned is the name of the class followed by the name of the method separated by double colon scope operator `::`. C++ requires the name of the class to distinguish this method from the method with the same name for a different concrete class. For example, the `Circle` class will also have a method named `area`. Its implementation will begin with the line

```
double Circle::area()
```

Within each method, the items in the private part of the class are available to use or to change. For example, the implementation of `area()`


```
Rectangle::Rectangle(double length, double width) {
    try {
        if (length < 0.0 || width < 0.0) {
            throw -1;
        }
        _length = length;
        _width = width;
    }
    catch(int a) {
        cerr << "Rectangle precondition violated: "
             << "length and width cannot be negative." << endl;
        exit(1);
    }
}
```

Figure 1.13 The C++ `try` / `throw` / `catch` statement.

```
return _length * _width;
```

uses the values of `_length` while the implementation of the constructor

```
_length = length;
```

changes the value of `_length`.

Preconditions are implemented with an `if` statement followed by a `throw` statement. For example, the precondition for the `Rectangle` constructor from Figure 1.11 is

```
// Pre: length >= 0.0 and width >= 0.0
```

which is implemented with the `if` statement to test if the precondition is false followed by the `throw` statement

```
throw -1;
```

in Figure 1.11. The effect of the throw statement is to abort the program if the boolean expression in its parameter list is false. The abort is accompanied with an error message that indicates what precondition was not satisfied. The precondition is a contract between the server, which is the class, and the client, which in this example is the main program that uses the class. This contract states that it is the responsibility of the client to insure that the parameters `length` and `width` are not negative when the constructor is called.

You may ask, Why put a statement in your program that will cause it to crash? Don't users hate programs that crash? The answer is that if the client program is written correctly, the boolean expression in the assert statement of the server program will never be false and the program will never crash. The assert statement is necessary not for the user but for the developer of the client program. When the client programmer is testing her code before releasing it commercially it is useful to have a controlled abort with an error message that pinpoints the cause of the error. Remember that it is frequently the

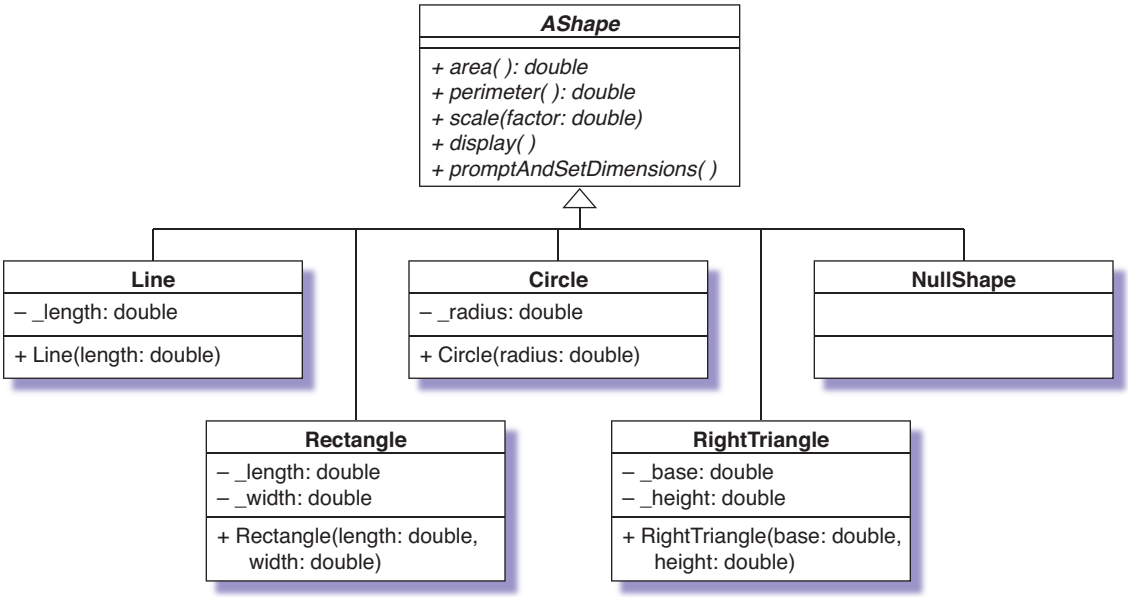


Figure 1.14 A class diagram for the abstract class `AShape` and the concrete classes `Line`, `Rectangle`, `Circle`, `RightTriangle`, and `NullShape`. Because each concrete class inherits and implements all the methods of the abstract class, they are not repeated in the operations part of the class box as they are in the class box of `Rectangle` in Figure 1.10. Shading is behind those classes that provide an implementation.

case that one programmer writes the server program while another one writes the client program.

If you write both the server program and the client program you may be able to keep them consistent without the checks enforced by the contract. However, even if a single individual does write both programs it is still beneficial to program with preconditions in the contract. The contract is at the boundary of a low level of abstraction, the server, and a high level of abstraction, the client. The precondition enforces the abstraction that relieves you of the burden of maintaining the details of the entire program in your mind. When you are programming the server, you do not need to think about how the client will insure that `length` is not negative. When you are programming the client, you do not need to think about how the client will create a new object, as long as you supply a nonnegative value for `length`.

C++ programs normally do not use the `throw` statement as in Figure 1.11. Instead, they use it in conjunction with the `try` and `catch` statements as part of an exception handling system. Figure 1.13 is an implementation of the function that does the same processing but using the `try` and `catch` statements with the `throw` statement.

In the `try` statement, if the precondition is true then the `throw` statement does not execute and the `catch` statement does not execute either. If the precondition is false then the `throw` statement executes and the remainder of the `try` statement is skipped. Execution of the `throw` triggers a search for a `catch` with a parameter whose type

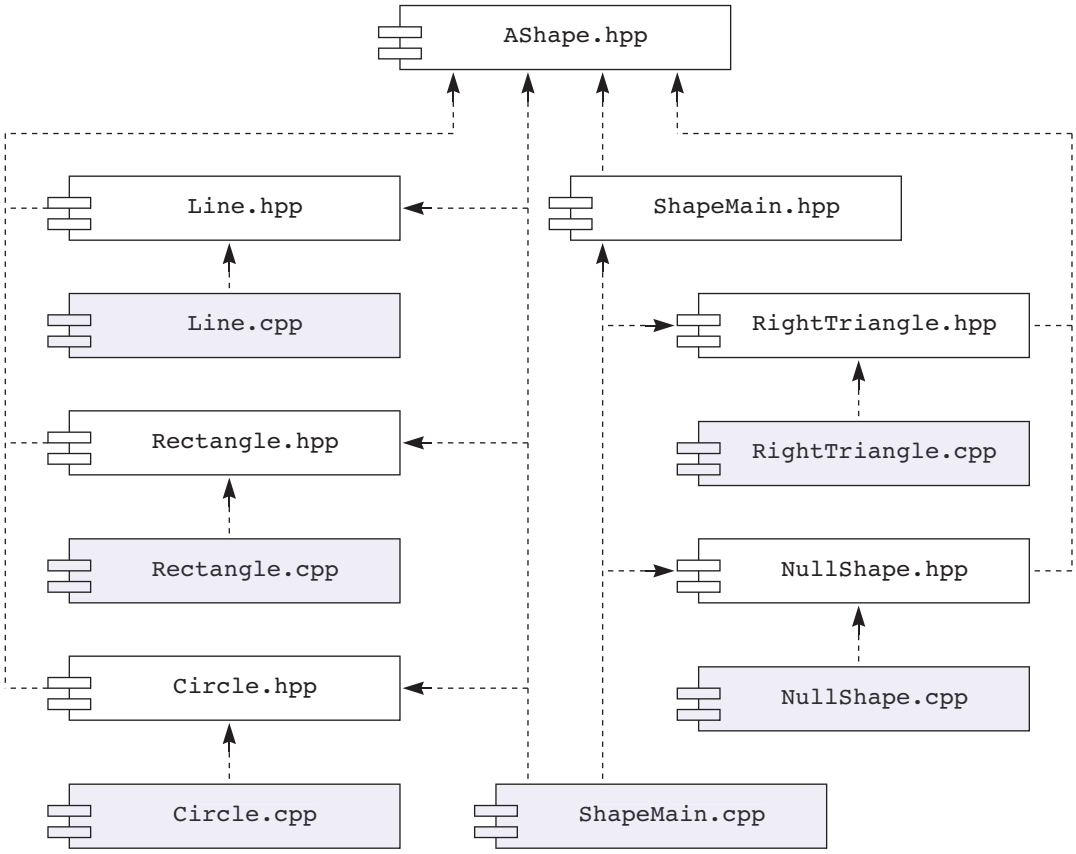


Figure 1.15 A UML component diagram of the compiler dependencies for the client program in `ShapeMain.cpp` that uses the `shape` classes. Some library files are not shown, such as the `iostream` header file for the `cout` statement. Also omitted is a utility file `Utilities.hpp` that provides the constant pi for the `circle` implementation and some routines to prompt the user for numeric input within specified limits. Those classes that provide an implementation are shaded.

matches the type thrown. In this case, the type of `a` is `int`, which matches the type of `-1`. Within the `catch` statement, the value of variable `a` is -1, which it gets from the `throw` statement. The `exit` statement aborts the program with an exit code of 1. By convention, an exit code of 0 indicated normal termination, and any nonzero value indicates abnormal termination.

Even this implementation is not a typical use of the exception handling system in C++. Many other intricate features exist that allow the programmer to catch an exception in one function that was thrown in another function. In Figure 1.11, there is no `try` or `catch`, so the search for a `catch` moves up to the function that called the `Rectangle` constructor. The search keeps moving up the call chain. If no `catch` is ever found, the program aborts. You can catch an exception at any point on the call chain. Such advanced techniques are beyond the scope of this book.


```
There are [0..4] shapes.
(m)ake (c)lear (a)rea (p)erimeter (s)cale (d)isplay (q)uit: m
Which shape? (0..4): 7
Must be between 0 and 4.
Which shape? (0..4): 2
(l)ine (r)ectangle (c)ircle right(t)riangle (m)ystery: r
Length? (>= 0): 2.5
Width? (>= 0): 3.0

There are [0..4] shapes.
(m)ake (c)lear (a)rea (p)erimeter (s)cale (d)isplay (q)uit: a
Which shape? (0..4): 2

Area: 7.5

There are [0..4] shapes.
(m)ake (c)lear (a)rea (p)erimeter (s)cale (d)isplay (q)uit: d
Which shape? (0..4): 2

Rectangle
Length: 2.5
Width: 3

There are [0..4] shapes.
(m)ake (c)lear (a)rea (p)erimeter (s)cale (d)isplay (q)uit: q
```

Figure 1.16 An interactive session of the main program that uses the abstract and concrete classes. There are five shapes numbered 0 through 4. For any of the five shapes, the user has a choice to make a new shape, clear the shape to the null shape, compute the area or length of the perimeter, scale the dimensions by a scale factor, or display the name and dimensions of the shape. User responses are bold.

Similar implementation code, not shown here, is necessary for each concrete shape. Figure 1.14 is a UML class diagram that shows the inheritance relationships between the concrete and abstract shapes. The concrete class named `NullShape` is included for the convenience of the client program. It has nothing in its private part, returns zero for its area and perimeter, and prints nothing for its name and dimensions. As described in the next section, having a null shape simplifies the code in the client program by eliminating the need to test for the existence of a shape. This design is an example of the object-oriented design pattern known as the *null object* pattern.

A client application

Figure 1.15 is a UML component diagram for a main program that uses the abstract and concrete shapes. The complete system requires the 13 files shown in the figure plus a utility file not shown. The implementation file for each concrete class depends on its

specification in the corresponding header file, which in turn depends on the header file containing the specification for the abstract class. The header file for the main program depends on the specification of the abstract shape. The implementation file for the main program depends on the specification of each concrete class and the specification of the abstract class as well as specifications in its own header file. Figure 1.16 shows a sample interactive session produced by the main program.

Figure 1.17 shows the header file for the main program. The parameters in all the parameter lists of the functions refer only to the abstract shape class, `AShape`. Nowhere in the header file is any reference to a concrete class such as `Rectangle`. Consequently, this header file depends only on `AShape.h`.

Each function is documented with a postcondition and possibly a precondition. Many of the functions have a side effect of prompting the user for some input, which is documented as well. It should be straightforward to compare the functions in Figure 1.17 with the interactive session in Figure 1.16 to determine which function is responsible for which prompt. For example, `shapeType()` produces the prompt

```
(l)ine  (r)ectangle  (c)ircle  right(t)riangle  ...
```

as a side effect. Its purpose is to return one of the uppercase letters `L`, `R`, `C`, `T`, or `M`. The user has no concept of the null shape, which is not one of the options. In addition to the line, rectangle, circle, and right triangle there is an option for a mystery shape. This option is provided for the student to complete as an exercise.

The functions in Figure 1.17 have three different types of formal parameters. The simplest parameter is `shared_ptr<AShape>` as in the function

```
void printArea(shared_ptr<AShape> sh);
```

The data type `shared_ptr` is the C++ notation for a shared pointer, and `<>` is the notation for a type. This declaration says that formal parameter `sh` is a pointer to the abstract class `AShape`. If you do not have experience with pointers in C++ or some other programming language, Chapter 2 describes pointer manipulations in detail. For now, it is sufficient to simply think of a pointer as an arrow pointing to an abstract shape, as in Figure 1.18(a).

The second type of parameter is `shared_ptr<AShape>&` as in the function

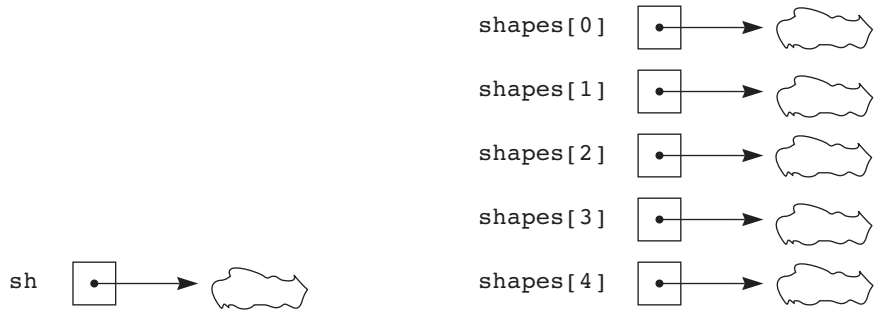
```
void makeShape(shared_ptr<AShape> &sh);
```

With this parameter, `sh` is also a pointer to an abstract type, as it is in the function `printArea()`. So, you can also visualize this parameter as in Figure 1.18(a). In `makeShape()`, however, `sh` is called by reference, indicated by the `&` symbol. The purpose of call by reference is to change the value of the actual parameter in the calling function. The purpose of function `makeShape()` is to prompt the user for a particular shape, then, depending on the shape, prompt for the desired dimensions. In Figure 1.16, the user requested a rectangle, and so was prompted for the rectangle's length and width. Then, function `makeShape()` changed the actual parameter to be a pointer to a rectangle having width 2.5 and height 3.0. Because the pointer changes to point to a different shape than it was pointing to before, the parameter must be called by reference.

The rule in C++ is that the absence of the `&` symbol implies call by value as the default. In call by value, the formal parameter gets the value of the actual parameter at the time of the call. Any changes that the called function makes to the formal parameter


```
#ifndef ShapeMain_hpp
#define ShapeMain_hpp

#include <memory>
#include "AShape.hpp"

void initialize(shared_ptr<AShape> shapes[], int cap);
// Pre: shapes[0..cap - 1] is allocated.
// Post: All shapes[0..cap - 1] are initialized to the null shape.

void promptLoop(shared_ptr<AShape> shapes[], int cap);
// Loop to prompt the user with the top-level main prompt.
// Post: User has selected the quit option.

void makeShape(shared_ptr<AShape> &sh);
// Prompts user for dimensions.
// Post: Original sh is deleted and new sh is created.

char shapeType();
// Prompts user for shape letter, lowercase or uppercase.
// Post: Uppercase character L, R, C, T, or M is returned.

void clearShape(shared_ptr<AShape> &sh);
// Post: sh is made the null shape.

void printArea(shared_ptr<AShape> sh);
// Post: sh's area is printed to standard output.

void printPerimeter(shared_ptr<AShape> sh);
// Post: The perimeter of this sh is printed to standard output.

void scaleShape(shared_ptr<AShape> sh);
// Prompts user for scale factor.
// Post: sh's dimensions are multiplied by the factor.

void displayShape(shared_ptr<AShape> sh);
// Post: sh's name and dimensions are printed to standard output.

#endif
```

Figure 1.17 `ShapeMain.hpp`. The header file for a main program that uses the shape classes. In Figure 1.15, the single dependency arrow pointing from `ShapeMain.hpp` to `AShape.hpp` corresponds to the `include` statement in this file that includes `AShape.hpp`.



(a) Visual depiction of parameter `sh`.

(b) Visual depiction of parameter `shapes`.

Figure 1.18 Visual depictions of the parameters of the functions in Figure 1.17. Each cloud represents one abstract shape as in Figure 1.7.

are made to the value copied at the time of the original call. Those changes are not reflected in the actual parameter. See Section A.6 in Appendix A and Section 2.1 in Chapter 2 for further discussions of the difference between call by reference and call by value.

The third type of parameter is `shared_ptr<AShape>` where the parameter name is followed by a pair of brackets `[]` as in

```
void initialize(shared_ptr<AShape> shapes[], int cap);
```

The brackets indicate that the formal parameter `shapes` is an array. Again, `shared_ptr<AShape>` indicates that it is an array of pointers to abstract shapes. Figure 1.16 shows that the user has a choice of five shapes, numbered 0 through 4. The five shapes are stored in the shapes array as Figure 1.18(b) shows.

You can see from the documentation of `initialize()`, that its purpose is to make all the shapes the null shape. Because each pointer must be changed to point to a null shape, shapes should be called by reference. So, why does the type in the parameter list not have the `&` symbol? Because in C++ the name of an array is different from the name of other variables. The value of an array is the address of its first element. Consequently, any time the formal parameter is an array, the effect is as if the array is called by reference, even without the `&`. In C++, you cannot pass the values of an array to the called function.

Figure 1.19, shows the listing of the main program and one function it calls. The main program is short, only five lines long. At the highest level of abstraction, the main program simply declares the array of five shapes and then uses function calls to initialize them and prompt for the user to manipulate them. The advantage of using type `shared_ptr` over the older "raw pointer" is that garbage collection is automatic. If the `Shape` project used raw pointers, the programmer would be responsible for deleting the object from the memory heap.

Function `initialize()` illustrates the cardinal rule of object-oriented assignment. Suppose you have an object named `base` instantiated from class `Base` declared as

```
Base base;
```

and another object `derived` instantiated from class `Derived` declared as


```
#include <cstdlib> // EXIT_SUCCESS.
#include <cctype> // toupper.

#include "Utilities.hpp" // promptIntBetween, promptDoubleGE.
#include "AShape.hpp"
#include "Line.hpp"
#include "Rectangle.hpp"
#include "Circle.hpp"
#include "RightTriangle.hpp"
#include "NullShape.hpp"
#include "ShapeMain.hpp"
#include "MysteryShape.hpp"

int main() {
    const int NUM_SHAPES = 5;
    shared_ptr<AShape> shapes[NUM_SHAPES];
    initialize(shapes, NUM_SHAPES);
    promptLoop(shapes, NUM_SHAPES);
    return EXIT_SUCCESS;
}

void initialize(shared_ptr<AShape> shapes[], int cap) {
    for (int i = 0; i < cap; i++) {
        shapes[i] = make_shared<NullShape>();
    }
}
```

Figure 1.19 `ShapeMain.cpp`. The main program that uses the abstract and concrete classes. Function `initialize` is shown here. The program continues in the next figure.

```
Derived derived;
```

where `Derived` inherits from `Base` as follows.

```
class Derived : public Base
```

The cardinal rule of object-oriented assignment says that *you can assign the specific to the general, but you cannot assign the general to the specific*. That is, the assignment

```
base = derived;
```

is legal, but the assignment

```
derived = base;
```

is not legal.

The class that is general contains items that are common to all the specific classes derived from it. A specific class inherits all those items and may contain additional

```
void promptLoop(shared_ptr<AShape> shapes[], int cap) {
    char response = '\0';
    do {
        cout << "\nThere are [0.." << cap - 1 << "] shapes." << endl;
        cout << "(m)ake  (c)lear  (a)rea  (p)erimeter  (s)cale  (d)isplay  "
             << "(q)uit: ";
        cin >> response;
        switch (toupper(response)) {
        case 'M':
            makeShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
            break;
        case 'C':
            clearShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
            break;
        case 'A':
            printArea(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
            break;
        case 'P':
            printPerimeter(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
            break;
        case 'S':
            scaleShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
            break;
        case 'D':
            displayShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
            break;
        case 'Q':
            break;
        default:
            cout << "\nIllegal command." << endl;
            break;
        }
    } while (toupper(response) != 'Q');
}
```

Figure 1.20 `ShapeMain.cpp` (continued). Function `promptLoop` from the main program listing. The main program listing continues in the next figure.

items. Because you can assign a specific object to a general object, you can endow a general object dynamically (that is, during execution of the program) with more items than were included in its original specification statically (that is, during compilation of the program). You can give, but you cannot take away. During execution, an object can get more specific than its static declaration, but it cannot get more general.

In function `initialize()`, formal parameter `shapes[]` is an array of pointers to `AShape`, the base class, which is general. An example of assignment of the specific to the general is when `initialize()` executes the statement


```
shapes[i] = make_shared<NullShape>();
```

This statement creates a specific object, a null shape, and assigns `shapes[i]` to point to the newly created null shape. Hence, `shapes[i]`, which is declared to be a pointer to a general shape statically, gets assigned to point to a specific shape dynamically.

This assignment also illustrates the operation of `make_shared<>()`. When the `make_shared<>()` function executes it

- allocates storage from the heap for the attributes of the object, and
- returns a pointer to the newly allocated storage.

In the above assignment statement, the `make_shared<NullShape>()` function call allocates storage from the heap for a null shape and returns a pointer to the null shape, which is given to `shapes[i]`.

Figure 1.20 shows the implementation of function `promptLoop()`. The function prompts the user with the main prompt, asking for some action to be executed. If the user, for example, types the letter `m`, the `switch` statement executes

```
makeShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
```

The index of `shapes` is

```
promptIntBetween("Which object?", 0, cap-1)
```

which prompts the user for an integer between 0 and 4. If, as in Figure 1.16, the user enters the erroneous value of 7 followed by the valid value of 2, `promptIntBetween` returns 2. The effect of the call is

```
makeShape(shapes[2]);
```

So, function `makeShape` is called with `shapes[2]` passed as the actual parameter.

Figure 1.21 shows the implementation of function `makeShape()`. Function `makeShape()` contains no arrays. Its formal parameter is a single pointer to an abstract shape named `sh`. Because the actual parameter is `shapes[2]`, and `sh` is called by reference, every change to `sh` in function `makeShape` is really being made on `shapes[2]`.

Function `makeShape()` executes the `switch` statement

```
switch (shapeType())
```

which in turn invokes function `shapeType()`, which then returns one of the characters `L`, `R`, `C`, `T`, or `M`. Strictly speaking, the default case is not necessary, nor is the last break statement in this switch. However, it is considered good C++ programming practice to always include them.

Function `makeShape()` illustrates the cardinal rule of object-oriented assignment. Formal parameter `sh` is a pointer to `AShape`, the base class, which is general. An example of assignment of the specific to the general is when the user wants to make a new rectangle. The statement

```
sh = make_shared<Rectangle>();
```

creates a specific object, a rectangle, and assigns `sh` to point to the newly created rectangle. Hence, `sh`, which is declared to be a pointer to a general shape statically, gets a pointer to a specific shape dynamically.


```
void makeShape(shared_ptr<AShape> &sh) {
    switch (shapeType()) {
    case 'L':
        sh = make_shared<Line>();
        break;
    case 'R':
        sh = make_shared<Rectangle>();
        break;
    case 'C':
        sh = make_shared<Circle>();
        break;
    case 'T':
        sh = make_shared<RightTriangle>();
        break;
    case 'M':
        //Exercise for the student.
        break;
    default:
        break;
    }
    sh->promptAndSetDimensions();
}

char shapeType() {
    char ch;
    cout << "(l)ine  (r)ectangle  (c)ircle  right(t)riangle  (m)ystery: ";
    cin >> ch;
    ch = toupper(ch);
    while (ch != 'L' && ch != 'R' && ch != 'C' && ch != 'T' && ch != 'M') {
        cout << "Must be l, r, c, t, or m.  Which type? ";
        cin >> ch;
        ch = toupper(ch);
    }
    return ch;
}
```

Figure 1.21 `ShapeMain.cpp` (continued). Functions `makeShape` and `shapeType` from the main program listing. The main program listing continues in the next figure.

This assignment also illustrates the implicit call to a constructor. The existence of a constructor for the `rectangle` class causes the `make_shared<>()` function to insert a call to the constructor after allocation from the heap. Compare the following steps to those listed on page 28 where the `null` shape class does not have a constructor. In this case, because the `Rectangle` class does have a constructor, when the `make_shared<>()` function executes it does three things. It

- allocates storage from the heap for the attributes of the object,
- calls the constructor based on the number and types of parameters in the parameter list, and
- returns a pointer to the newly allocated storage.

In the above assignment statement, first the `make_shared<>()` function allocates storage from the heap for the attributes of `Rectangle`, which, from Figure 1.9 are

```
double _length;
double _width;
```

Second, the `make_shared<>()` function calls the constructor based on the number and types of parameters. Because the actual parameter list is empty, and the constructor for `Rectangle` in Figure 1.9 has two parameters `length` and `width` with default values, the constructor is called with both parameters having their default values of 0.0. Figure 1.11 shows the implementation of the constructor, which does the assignment

```
_length = length;
_width  = width;
```

giving 0.0 to the attributes of the shape object. Third, the `make_shared<>()` function returns a pointer to newly allocated and initialized storage.

All that happens on the right hand side of the assignment statement

```
sh = make_shared<Rectangle>();
```

The assignment completes by giving `sh` the pointer to the rectangle. After `break` executes in the `switch` statement, the statement

```
sh->promptAndSetDimensions();
```

executes. What happens now is "polymorphic dispatch," the topic of the next section.

1.3 Polymorphism

The highest level of abstraction in object-oriented programming languages is behavior abstraction, which is manifested in polymorphism. This section concludes the discussion of the abstract shape main program, which illustrates polymorphism.

Behavior abstraction

The behavior of a program is determined by the sequential execution of consecutive statements, selection of various optional sections of code with some form of `if` or `switch` statement, and repetition with some form of `while` or `for` statement. The essence of abstraction is the hiding of detail, so behavior abstraction implies the hiding of one of these control mechanisms, namely `selection`. With polymorphism you can eliminate `if` or `switch` statements from your code where they would be required without polymorphism. Consequently, the control structure for objects is simplified by the abstraction process. The selection of an alternative without the usual `if` or `switch` statement is known as *polymorphic dispatch*.

Consider the statement


```
sh->promptAndSetDimensions();
```

from function `makeShape()` in Figure 1.21. The symbol `->` is the C++ operator for accessing the field of a `struct` or `class` when the left hand side is a pointer to the `struct` or `class`. This statement is located after a `switch` statement, so `sh` could be a pointer to any one of a number of shapes. The scenario considered at the end of the previous section assumed that the user selected a rectangle, but he could just have easily picked a circle or triangle. Furthermore, the static type of `sh` is `AShape`, which is general. Think what the compiler must do to translate this statement. Should the compiler generate machine language statements to call `promptAndSetDimensions()` for a rectangle? Or should it generate statements to call the corresponding function for a circle or triangle? It cannot generate statements to call `promptAndSetDimensions()` for `AShape`, because `AShape` has no implementation of that function. There are no implementations of methods for `AShape`, only their pure virtual specifications in Figure 1.8.

The answer is that the compiler generates code that, in effect, tests the *dynamic type* of `sh` and calls the corresponding version of `promptAndSetDimensions()`. During execution, if the dynamic type of `sh` is `Rectangle` then `promptAndSetDimensions()` for `Rectangle` will be called. If the dynamic type of `sh` is `Circle` then `promptAndSetDimensions()` for `Circle` will be called. The machine language code that tests the dynamic type of `sh` and calls the appropriate version of the method is hidden at a lower level of abstraction. It does not appear in the C++ program.

As another example of polymorphic dispatch, consider Figure 1.22, which shows the implementation of `printArea()`, `printPerimeter()`, `scaleShape()`, and `display()`. For example, the implementation of function `printArea()` is

```
void printArea(shared_ptr<AShape> sh) {
    cout << "Area: " << sh->area() << endl;
}
```

The purpose of the function is to print the name and dimensions of the shape to the standard output stream. How does the function know what to print? If the shape is a rectangle, the function should print the string "Area" followed by the product of its width and height, but if the shape is a circle it should print the string "Area" followed by pi times the square of its radius.

Consider how you would implement this function without polymorphism. You would need a way to distinguish what kind of shape `sh` is. For example, you might declare your type `Shape` as follows.

```
enum ShapeKind {
   E_LINE, E_RECTANGLE, E_CIRCLE,
   E_RIGHT_TRIANGLE, E_NULL_SHAPE
};
typedef ShapeType {
   ShapeKind kind;
   double dim1, dim2;
};
```

Field `kind` in the definition of the type could be used to distinguish what kind of shape is stored, with the integers interpreted accordingly. For example, if `kind` had value


```
void clearShape(shared_ptr<AShape> &sh) {
    sh = make_shared<NullShape>() ;
}

void printArea(shared_ptr<AShape> sh) {
    cout << "\nArea: " << sh->area() << endl;
}

void printPerimeter(shared_ptr<AShape> sh) {
    cout << "\nPerimeter: " << sh->perimeter() << endl;
}

void scaleShape(shared_ptr<AShape> sh) {
    sh->scale(promptDoubleGE("Scale factor?", 0.0));
}

void displayShape(shared_ptr<AShape> sh) {
    cout << endl;
    sh->display();
}
```

Figure 1.22 `ShapeMain.cpp` (continued). Functions `printArea()`, `printPerimeter()`, `scaleShape()`, and `displayShape()` exhibit polymorphic dispatch. This concludes the main program listing.

`E_RECTANGLE` then `dim1` would store the length of the rectangle and `dim2` would store its width. But if kind had value `E_CIRCLE` then `dim1` would store the circle's radius and `dim2` would be ignored.

In your implementation of `printArea()`, you would need to test the `kind` field to determine which shape is stored in `sh` as follows.

```
void printArea(shared_ptr<AShape> sh) {
    switch (sh->kind) {
    case E_LINE:
        cout << "Area: " << lineArea(sh) << endl;
        break;
    case E_RECTANGLE:
        cout << "Area: " << rectangleArea(sh) << endl;
        break;
    etc.
    }
}
```

Each function would return the area of the corresponding shape. For example, function `rectangleArea()` would be implemented as

```
double rectangleArea(shared_ptr<AShape> sh) {
```

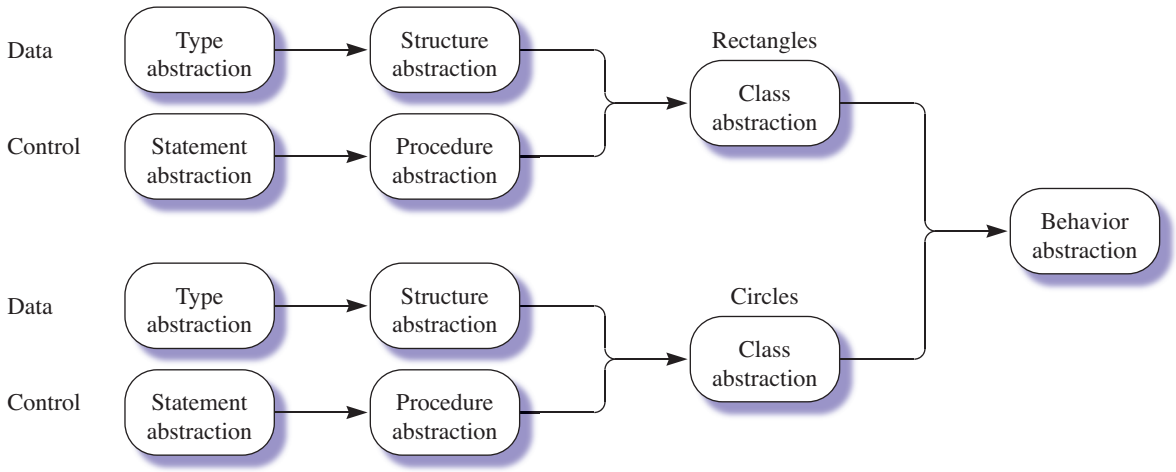


Figure 1.23 The abstraction process. Each step in the process consists of collecting many items at one layer into a single concept at the next higher layer. Type and structure abstraction are aspects of data abstraction, and statement and procedure abstraction are aspects of control abstraction.

```
    return sh->dim1 * sh->dim2
}
```

Compare this with the corresponding implementation of the function in Figure 1.11 to compute the area of a rectangle object, which is an instantiation of a subclass of the abstract `AShape` class.

```
double Rectangle::area() {
    return _length * _width;
}
```

Both functions perform the same computation. The difference is how they are called.

Which brings us back to our original question, How does the `printArea()` function know what to print? It simply calls the function `sh->area()`. After all, there is an `area()` method for each shape. Furthermore, the compiler cannot detect from the type of `sh` which method should be called, because `sh` is an abstract shape, and it could be any specific shape at execution time.

Again, the answer is polymorphic dispatch. At a lower level of abstraction, invisible to the programmer at the C++ level, the compiler includes a tag field, much like the field `kind` in the above description, with every instantiation of a subclass. Also, when the compiler translates the function call `sh->area()` it uses the tag field to look up in a table (called a virtual method table) the appropriate function to call. Even though the compiler does not know at compile time which method will be called, it generates code to make that determination at execution time. The programmer only needs to make the function call `sh->area()`. It is as if the `sh` object knows what shape it is and returns its area without the programmer needing to test what its shape is.

Behavior abstraction with polymorphism is the highest level of abstraction within object-oriented languages such as C++. In the same way that class abstraction brings together attributes and operations into a single class, behavior abstraction brings together a collection of different classes under the umbrella of an abstract class. At the highest level, the programmer writes code to the abstraction provided by the abstract class and lets polymorphic dispatch automatically take care of the details at a lower level of abstraction. Figure 1.23 shows the progression of the abstraction process.

Reusability and extensibility

[*This section will describe the benefits of OO design using the Shapes example as a point of departure*.]

Design patterns

The first question to ask when solving a problem is, What is the problem? Before you can write a program to solve a problem you must understand the specification of the problem. Abstraction is a tool that helps you determine the specification of the problem at a high level. At this level of abstraction many details are hidden with only the essentials exposed.

To solve a problem you must implement its specification. An implementation requires you to structure your data to create a model of the abstract view. The narrower the gap between the structure of the data and the abstract view of the problem, the easier it is to write a correct solution. A narrow gap between the abstract view and the implementation usually produces a more elegant solution as well.

Not only can you design an abstraction boundary between the specification of a problem and its implementation, you can also design layers of abstraction within the implementation. Within one of the layers of abstraction you can subdivide the problem into a system of cooperating objects to further reduce complexity. Without layers of abstraction, a problem solution becomes one large monolithic program that is difficult to understand and debug. Layering structures a complex problem into more manageable parts that are then easier to understand and construct.

[*This section will describe the idea of OO design patterns in the context of abstraction*.]

1.4 The Factory Pattern

[*This section will describe the ShapeFactory project*]

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

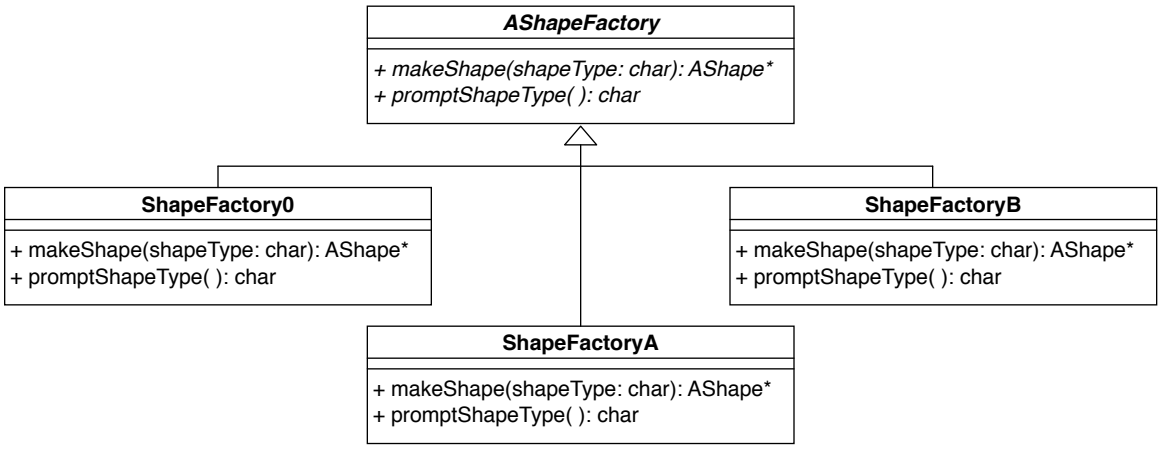


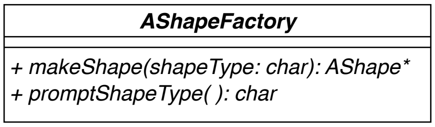
Figure 1.24 A class diagram for the abstract class `AShapeFactory` and the concrete classes `AShapeFactory0`, `AShapeFactoryA`, and `AShapeFactoryB`.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information



```
class AShapeFactory {
public:

    virtual ~AShapeFactory() { }
    // Virtual destructor necessary for subclassing.

    virtual AShape* makeShape(char shapeType = 'N') = 0;
    // Post: A pointer to a shape of type shapeType is returned.

    virtual char promptShapeType() = 0;
    // Prompts user for shape letter, lowercase or uppercase.
    // Post: Uppercase character representing a shape type is returned.
};
```

Figure 1.25 `AShapeFactory.hpp`. Declaration of the abstract factory class of Figure 1.24. The box is the UML class diagram corresponding to the C++ code.

about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed

```
class ShapeFactory0 : public AShapeFactory{
public:
    AShape* makeShape(char shapeType = 'N') override;
    // Post: A pointer to a NullShape object is returned.

    char promptShapeType() override;
    // Does nothing.
    // Post: Null character is returned.
};
```

ShapeFactory0
+ makeShape(shapeType: char): AShape* + promptShapeType( ): char

(a) `ShapeFactory0.hpp`. Specification of `ShapeFactory0`.

```
AShape* ShapeFactory0::makeShape(char /*not used*/) {
    return new NullShape;
}

char ShapeFactory0::promptShapeType() {
    return '\0';
}
```

(b) `ShapeFactory0.cpp`. Implementation of `ShapeFactory0`.

Figure 1.26 `ShapeFactory0.hpp` and ShapeFactory0.cpp. The box is the UML class diagram corresponding to the C++ code.

text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original


```
class ShapeFactoryB : public AShapeFactory{
public:
    AShape* makeShape(char shapeType = 'N') override;
    // Post: A pointer to a shape of type shapeType is returned.
    // shapeType E for Ellipse, T for Equilateral Triangle,
    // and M for Mystery shape.

    char promptShapeType() override;
    // Prompts user for shape letter, lowercase or uppercase.
    // Post: Uppercase character E, T, or M is returned.
};
```

(a) `ShapeFactoryB.hpp`. Specification of `ShapeFactoryB`.

```
AShape* ShapeFactoryB::makeShape(char shapeType) {
    switch (shapeType) {
    case 'E':
        return new Ellipse;
    case 'T':
        return new EquilateralTriangle;
    case 'M':
        // Exercise for the student.
        break;
    default:
        return new NullShape;
    }
}

char ShapeFactoryB::promptShapeType() {
    char ch = '\0';
    cout << "(e)llipse  equilateral(t)riangle  (m)ystery: ";
    cin >> ch;
    ch = toupper(ch);
    while (ch != 'E' && ch != 'T' && ch != 'M') {
        cout << "Must be e, t, or m.  Which type? ";
        cin >> ch;
        ch = toupper(ch);
    }
    return ch;
}
```

ShapeFactoryB
+ makeShape(shapeType: char): AShape* + promptShapeType( ): char

(b) `ShapeFactoryB.cpp`. Implementation of `ShapeFactoryB`.

Figure 1.27 `ShapeFactoryB.hpp` and ShapeFactoryB.cpp. The box is the UML class diagram corresponding to the C++ code.

language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original

language. There is no need for special content, but the length of words should match the language.

Exercises

This book comes with a set of software called the "dp4ds distribution," to which you should have access for study and for working exercises. Exercises at the end of the chapter ask you to implement or modify parts of the distribution software. The distribution software is available at

`http://www.cslab.pepperdine.edu/warford/cosc320/`

1–1 Implement the code for the `scale()` method for the shapes `Line`, `Rectangle`, and `Circle`. Include the precondition test in your code.

1–2 Implement the code for all the methods including `scale()` for the shape `RightTriangle`. Include the precondition tests in your code.

1–3 Choose a shape other than a line, rectangle, circle or right triangle and implement all the methods for it including `scale()` as the mystery shape in Figure 1.16. Include the precondition tests in your code.

Chapter 2

Arrays and Pointers

C++ is a large and complex language, in part because of some of its design goals. One goal that contributes to its complexity is the requirement that C++ be backward compatible with C. That is, any program written in C should compile on a C++ compiler. The C language was originally designed to write the Unix operating system. That goal required the language to have low-level access to hardware features and to be efficient.

The advantage of backward compatibility is that programmers who know the C language can use the C++ language to gradually learn the object-oriented features of the newer language. This advantage is a primary reason for the widespread adoption of the language. The disadvantage is that the low-level features of C must be compatible with C++, which makes some of the capabilities of C++ difficult to use.

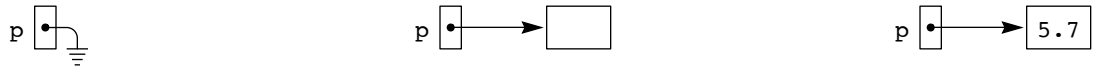
Early versions of C++ provide raw pointers that behave like pointers in C. With C pointers and C++ raw pointers, it is easy to make errors with memory deallocation that are difficult to detect with casual testing. Specifically, if a program fails to correctly delete all the unused storage from the heap, unused memory cells will accumulate in the heap and will not be available with later attempts to allocate from the heap. Such a bug is called a *memory leak*. It is possible that after some period of time the entire heap will be occupied with unused cells, but allocation from the heap will nevertheless be impossible. When that happens, at best the program will cease operation with an error message warning the user that the program is out of memory, and at worse will crash. Memory leaks are notoriously difficult to detect, because the symptoms of the error are usually not manifest with short test runs of the program.

2.1 Pointer Types

A shared pointer is a class that contains, as one of its attributes, a raw pointer. This section shows how to allocate storage from the heap with shared pointers and how to manipulate them.

Shared pointers

Shared pointers alleviate the problem of memory leaks. The programmer is no longer responsible for deallocating storage from the heap with the `delete` operation. Instead,



(a) `shared_ptr<double> p;` **(b)** `p = make_shared<double>();` **(c)** `*p = 5.7;`

Figure 2.1 Allocating storage for a pointer to a double precision real, and assigning a real value to the allocated cell. The dashed triangle represents the `nullptr` value.

the execution system uses a reference counting algorithm that keeps track of how many pointers point to a shared object. When the number of references to the object decreases to zero so that the shared object can no longer be accessed, the system automatically deletes the object from the heap.

A shared pointer is an object of the standard class `shared_ptr`, which contains a raw pointer as an attribute. The non-member function `make_shared<>()` creates a shared pointer and takes the place of the `new` operation for raw pointers. The dereferencing operators `*` and `->` are provided as member functions of the class `shared_ptr` so you can access dynamically allocated storage with shared pointers just like you can with raw pointers. Here is an example of how to declare a shared pointer with the equivalent operations for a raw pointer.

```
shared_ptr<double> p;                double *p;
p = make_shared<double>();           p = new double;
*p = 5.7;                            *p = 5.7;
```

The `shared_ptr` type is used in C++ to declare a shared pointer variable. The statement

```
shared_ptr<double> p;
```

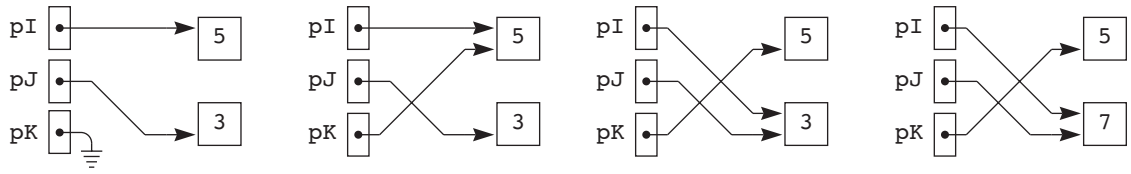
declares the variable `p` to be a pointer to a double precision value. Figure 2.1(a) shows the effect of the declaration.

Pointers are memory addresses, and at a low level of abstraction a memory address is an unsigned integer. In the C and C++ languages, the pointer value 0 is reserved as a special sentinel value because a cell can never be allocated from the heap at address 0. It is now considered good practice to use an alternative representation for 0 called `nullptr`, which is a keyword introduced in C++11. The use of `nullptr` in place of 0 makes the code easier to read because it shows clearly that the value is to be considered a pointer instead of an integer. Figure 2.1(a) shows the state of `p` after its declaration as being a pointer attached to a dashed triangle, which represents the special `nullptr` value. Smart pointers are automatically initialized to `nullptr`.

Before you can store a double precision value in the variable you must allocate storage for it from the heap. `make_shared<>()` is a function that allocates storage from the heap. It expects a type `T` in the angle brackets `<>`, allocates enough storage for `T`, and returns a pointer to the newly allocated storage. The statement

```
p = make_shared<double>();
```

allocates storage from the heap for a double precision value and returns a pointer to it. Variable `p` gets the value. Figure 2.1(b) shows the effect of the memory allocation. You can combine the declaration and allocation as follows.



(a) `*pI = 5;`
`*pJ = 3;`

(b) `pK = pI;`

(c) `pI = pJ;`

(d) `*pI = 2 + *pK;`

Figure 2.2 Allocating storage for pointers to integers and performing integer and pointer assignments.

```
shared_ptr<double> p = make_shared<double>();
```

The `*` operator is used to dereference a pointer. When placed before a pointer variable, the notation indicates the cell to which the pointer points. For example, the statement

```
*p = 5.7;
```

assigns the value 5.7 to the memory cell to which `p` points, as in Figure 2.1(c). From the hardware point of view, `p` is the memory address of the location where the value is stored, and `*p` is the memory location itself.

You can combine the declaration, allocation, and initialization all in one statement as follows.

```
shared_ptr<double> p = make_shared<double>(5.7);
```

The function `make_shared<>()` has an optional parameter for initializing the value to which the pointer points.

You can assign one pointer to another, but you must be careful to consider the effect of such an assignment. Because a pointer "points to" an item, if you give the pointer's value to a second pointer, the second pointer will point to the same item to which the first pointer points. Consider the following code fragment, illustrated in Figure 2.2.

```
shared_ptr<int> pI = make_shared<int>();
shared_ptr<int> pJ = make_shared<int>();
shared_ptr<int> pK;
*pI = 5;
*pJ = 3;
pK = pI;
pI = pJ;
*pI = 2 + *pK;
cout << *pI << " " << *pJ << " " << *pK << endl;
```

The assignment of `pI` to `pK` is a pointer assignment, not an integer assignment. It makes `pK` point to the same cell to which `pI` points as shown in Figure 2.2(b). There is no change of any cell content. Similarly, the assignment of `pJ` to `pI` makes `pI` point to the same cell to which `pJ` points as in Figure 2.2(c). Execution of the output statement streams

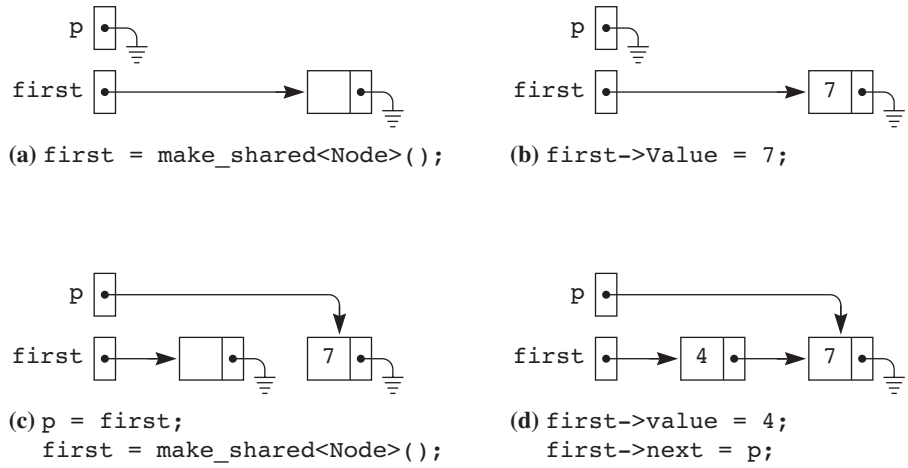


Figure 2.3 Trace of a code fragment that uses the `->` operator.

```
7 7 5
```

to `cout`. Because `pI` and `pJ` now point to the same memory cell, the cell containing 7, its value is printed twice.

In practice, pointers rarely point to primitive types like `double` and `int`. Instead, they point to a `struct` or a `class`. An example is a node structure for a linked list declared as

```
struct Node {
    int value;
    shared_ptr<Node> next;
};
shared_ptr<Node> first, p;
```

Variable `p` has type pointer to `Node`. Therefore, *p has type `Node`, which is a struct. You access a field of a `struct` or a `class` with the period operator `.` which is placed between the reference to the `struct` and its field. Thus, `(*p).value` is the value field of the `struct` to which `p` points, and `(*p).next` is the `next` field of the `struct`. The parentheses are necessary because the period operator has higher precedence than the `*` operator as Figure A.1 of the Appendix shows. Fortunately, C++ provides the `->` operator, which allows the programmer to combine the `*` operator and the period operator without using parentheses. For example,

```
p->value
```

is equivalent to

```
(*p).value
```

Figure 2.3 is a trace of the execution of the following code fragment, which uses the `->` operator.

```
first = make_shared<Node>();
first->value = 7;
p = first;
first = make_shared<Node>();
first->value = 4;
first->next = p;
for (shared_ptr<Node> q = first; q != nullptr; q = q-> next) {
    cout << q -> value << " ";
}
```

The `for` loop outputs

```
4 7
```

The loop can be simplified to

```
for (shared_ptr<Node> q = first; q; q = q-> next) {
```

which is also a common coding pattern in C. The value `nullptr` is interpreted as `false`, and so `q` is interpreted as `true` when it has any value other than `nullptr`.

Placeholder type specifiers

C++11 introduced placeholder type specifiers, also known as automatic type deduction. When you declare a variable with an initializer you can omit the type of the variable and replace it with the keyword `auto`. The compiler will then automatically deduce the type of the variable from the initializer.

For example, instead of declaring `i` explicitly to be an integer as

```
int i = 7;
```

replace `int` with `auto` as

```
auto i = 7;
```

The compiler will deduce that the type of `i` is `int` from the constant `7` because it does not contain a decimal point. Here are some type deductions provided by `auto`.

```
auto i = 7;                                int
auto x = 6.2;                              double
auto p = make_shared<double>(5.7);         shared_ptr<double>
auto pI = make_shared<int>();              shared_ptr<int>
auto first = make_shared<Node>();          shared_ptr<Node>
for (auto q = first; q; q = q-> next)      shared_ptr<Node>
```

The benefit of using `auto` may not seem like much in these simple examples. However, when dealing with complex data structures with involved types using `auto` as an abbreviation can shorten and simplify your code.


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

void swapVal(int g, int h) {
    auto temp = g;
    g = h;
    h = temp;
    cout << "g == " << g << ", h == " << h << endl;
}

int main() {
    auto i = 4;
    auto j = 5;
    swapVal(i, j);
    cout << "i == " << i << ", j == " << j << endl;
    return EXIT_SUCCESS;
}
```

Output

```
g == 5, h == 4
i == 4, j == 5
```

Figure 2.4 `SwapValMain.cpp`. A program to illustrate call by value. The formal parameters are `g` and `h`, which correspond to actual parameters `i` and `j`. Although the formal parameters change in the function `swapValue`, the actual parameters do not change in `main`.

Parameters

The Appendix describes the three parameter passing mechanisms of C++:

- Pass by value
- Pass by reference
- Pass by constant reference

The general rule is to use pass by reference when you want the function to change the value of the actual parameter, and to use the others when you do not. Following are example programs that show the ramifications of using the different passing mechanisms in a function.

Figure 2.4 shows the effect of passing parameters by value. When the function call executes, the processor copies the value of the actual parameter onto the run-time stack. During execution of the called function, any changes that it makes to the formal parameters it makes to the copies. The actual parameters in the calling function are not affected by the changes, because the changes are made to the copies, not to the original actual parameters. The output in the figure shows that `swapVal()` changes the values of formal parameters `g` and `h` but not the actual parameters `i` and `j`.


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

void swapRef(int &g, int &h) {
    auto temp = g;
    g = h;
    h = temp;
    cout << "g == " << g << ", h == " << h << endl;
}

int main() {
    auto i = 4;
    auto j = 5;
    swapRef(i, j);
    cout << "i == " << i << ", j == " << j << endl;
    return EXIT_SUCCESS;
}
```

Output

```
g == 5, h == 4
i == 5, j == 4
```

Figure 2.5 `SwapRefMain.cpp`. A program to illustrate call by reference. The call to `swapRef` passes an integer as the actual parameter with a reference type as the formal parameter. Unlike Figure 2.4 the value of actual parameter `i` is not passed. Rather, the address of `i` is passed.

Figure 2.5 shows call by reference. The formal parameters of `swapRef()` are `g` and `h`, each one of which is prefixed by `&` indicating call by reference. The actual parameters, `i` and `j`, are declared to be integers. When the function call executes, the processor passes a reference to the formal parameters. You can think of `g` as being a reference to `i` and `h` as being a reference to `j`. So, in the function when

```
h = temp;
```

executes, it is as if

```
j = temp;
```

executes.

Pointer parameters

The program in Figure 2.6 achieves the effect of pass by reference by explicitly passing pointers by value. The formal parameters `g` and `h` are pointers to integers called by value. Because they are called by value, the pointers themselves cannot change. However, the values in the cells to which they point can change.


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

void swapPtrVal(shared_ptr<int> g, shared_ptr<int> h) {
    auto temp = *g; // before
    *g = *h;
    *h = temp;
    cout << "*g == " << *g << ", *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrVal(i, j);
    cout << "*i == " << *i << ", *j == " << *j << endl;
    return EXIT_SUCCESS;
}
```

Output

```
*g == 5, *h == 4
*i == 5, *j == 4
```

(a) Passing pointers by value, before. **(b)** Passing pointers by value, after.

Figure 2.6 `SwapPtrValMain.cpp`. A program to illustrate passing pointers by value. This technique achieves the effect of call by reference by changing the values to which `i` and `j` point.

Corresponding to the formal parameters, the actual parameters in the call to the function are `i` and `j`, also pointers to integers, and initialized to 4 and 5 respectively. Figure 2.6(a) shows the memory allocation immediately after the function is called. Actual parameter `i` and formal parameter `g` are pointers with the same value. That is, they point to the same memory cell in the heap.

Figure 2.6(b) shows the memory allocation immediately before the return from the function. The code in the function has changed the values in the cells to which the pointers point.

Behind the scenes, the C++ compiler generates code like that of Figure 2.6 to implement call by reference when programmers use the technique of Figure 2.5. In Figure 2.5 the actual parameters are not pointers, but integers. So, the compiler passes the addresses of the integers. Any assignment to a formal parameter in the function, like


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

void swapPtrRef(shared_ptr<int> &g, shared_ptr<int> &h) {
    auto temp = g; // before
    g = h;
    h = temp;
    cout << "*g == " << *g << ", *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrRef(i, j);
    cout << "*i == " << *i << ", *j == " << *j << endl;
    return EXIT_SUCCESS;
}
```

Output

```
*g == 5, *h == 4
*i == 5, *j == 4
```

(a) Passing pointers by reference, before. **(b)** Passing pointers by reference, after.

Figure 2.7 `SwapPtrRefMain.cpp`. A program to illustrate passing pointers by reference. This technique achieves the effect of call by reference by changing what `i` and `j` point to.

```
g = h;
```

generates code to assign the value of `h` to the value of `g`. At the lowest level of abstraction every parameter is passed by value. The only question is, What kind of value is passed, a data value or an address value? The processor passes a data value to implement pass by value and an address value to implement pass by reference.

In Figure 2.7, the pointer itself changes what it points to, instead of changing the value in the cell to which it points. The pointer parameters are called by reference. Figure 2.7(a) shows the memory allocation immediately after the function is called. Actual parameter `i` and formal parameter `g` are, in effect, the same pointer. When `i` is changed in the function, `g` is changed. Figure 2.7(b) shows the memory allocation immediately before the return from the function. The code in the function has changed the pointers instead of changing the content of the cells to which they point. If you delete the `&`

symbols in the parameter list, the output will be

```
*g == 5, *h == 4
*i == 4, *j == 5
```

because the pointers will not be changed in the main program.

Reference types

In the function of Figure 2.5,

```
void swapRef(int &g, int &h) {
```

which is called as follows

```
auto i = 4;
auto j = 5;
swapRef(i, j);
```

`int &` is known as a *reference type*.

Reference types are not limited to function parameters. You can declare a reference type outside a parameter list, but if you do so you must initialize it when you declare it. For example, this code fragment

```
auto i = 4;
int &g = i; // Must be initialized here.
cout << "g == " << g << ", i == " << i << endl;
g = 5;
cout << "g == " << g << ", i == " << i << endl;
```

is legal and produces the output

```
g == 4, i == 4
g == 5, i == 5
```

Because `g` refers to `i`, when you change the value of `g` you also change the value of `i`. This behavior is identical to that in function `swapRef()`. When you change the value of formal parameter `g` in the function, you change the value of actual parameter `i`.

Once you set the reference of a reference type when you create it, you can never change what it refers to. If you have another integer variable `s` as in the following code fragment

```
auto i = 4;
int &g = i; // Must be initialized here.
auto s = 6;
g = s;
s = 7;
cout << "g == " << g << ", i == " << i << ", s == " << s << endl;
```

the assignment of `s` to `g` does not make `g` refer to `s`. Instead, it gives the value of `s` to `g`. The output is

```
g == 6, i == 6, s == 7
```



```
#include <iostream> // istream, ostream.
using namespace std;

void readStream(istream &is, unique_ptr<double[]> *d, int cap, int &len);
// Pre: d is allocated with capacity cap.
// Post: num values are input from is to d[0..num - 1], where
// num == min(number of elements in is, cap).
// len == num.

void writeStream(ostream &os, unique_ptr<double[]> const *d, int cap, int len);
// Pre: d is allocated with capacity cap.
// Post: num values are output from d[0..num - 1] to os, where
// num == min(len, cap).
```

Figure 2.8 `ArrayClassicMain.hpp`. The specification of a main program that uses a dynamically allocated array of doubles.

Changing the value of `s` to 7 does not change the value of `g`, which still refers to `i`.

This behavior is consistent with reference types in parameter lists. Think of a function call as creating a reference type on the run-time stack and initializing it the same way that `g` is initialized to refer to `i` in the above code fragments. You use the formal parameter in a function the same way you use the reference variable `g` in the above code. Once you set which cell a call-by-reference formal parameter refers to when the function is called, you can never change it to refer to a different cell in memory.

Dynamically allocated arrays

In Chapter 1, Figure 1.19 (page 26) shows the allocation of an array of pointers as

```
const int NUM_SHAPES = 5;
shared_ptr<AShape> shapes[NUM_SHAPES];
```

This allocation is static. That is, the array of pointers is allocated on the runtime stack of the main program. It is true that the objects to which each pointer points in Figure 1.18(b) (page 25) are allocated from the heap. But the array itself is allocated statically on the runtime stack.

A disadvantage of allocating an array statically is that the number of elements must be known at compile time. That is, `NUM_SHAPES` in the above declaration must be a constant. It cannot be a variable. If you want to change the number of shapes to process in the program, you must change the constant with your text editor and recompile the program. With static arrays, you cannot prompt the user for the size of the array as that would require a variable in the declaration of the array.

With dynamic allocation, the program allocates the array itself from the heap. Figure 2.8 shows the header file for a main program that uses dynamic allocation of an array of doubles, and Figure 2.9 shows its implementation. The main program prompts the user for the capacity of the array, which it stores in the integer variable `cap`. The statement

```
auto arr = make_unique<double[]>(cap);
```



```
#include <cstdlib> // EXIT_SUCCESS.
#include <fstream> // ifstream.
#include <memory>
#include "ArrayClassicMain.hpp"
#include "Utilities.hpp" // promptIntGE.
using namespace std;

int main() {
    int cap = promptIntGE("Enter array capacity", 1);
    auto arr = make_unique<double[]>(cap);
    ifstream ifs;
    promptFileOpen(ifs);
    if (ifs) {
        int length = 0;
        readStream(ifs, &arr, cap, length);
        ifs.close();
        cout << "Read count == " << length << endl;
        cout << "Array data:" << endl;
        writeStream(cout, &arr, cap, length);
        // arr[2 * cap] = 123.4;
        // cout << arr[2 * cap] << endl;
    }
    return EXIT_SUCCESS;
}

void readStream(istream &is, unique_ptr<double[]> *d, int cap, int &len) {
    len = 0;
    for (int i = 0; i < cap && is >> (*d)[i]; i++) {
        len++;
    }
}

void writeStream(ostream &os, unique_ptr<double[]> const *d, int cap, int len) {
    for (int i = 0; i < len && i < cap; i++) {
        os.width(12);
        os << (*d)[i];
        if (i % 6 == 5) {
            os << endl;
        }
    }
    os << endl;
}
```

Figure 2.9 `ArrayClassicMain.cpp`. The implementation of the main program specified in Figure 2.8.

allocates the array dynamically from the heap. Allocation from the heap allows `cap` to be a variable.

The C++20 standard provides the ability to have a shared pointer to an array. Unfortunately, at the time of this writing C++20 compilers are not widely available. So, the dp4ds Distribution is designed for standard C++17, which does not allow shared pointers to arrays. The `unique_ptr` type is for pointers to objects that are not shared and provides the same automatic garbage collection the `shared_ptr` type provides. The `make_unique<>()` funcation is for dynamic allocation for unique pointers corresponding to the `make_shared<>()` function for shared pointers.

This restriction to C++17 requires the array parameters in functions `readStream()` and `writeStream()` to be more complicated than would otherwise be the case. The formal parameter `unique_ptr<double[]> *d` is a raw pointer to a unique pointer to an array. The actual parameter `&arr` is the address of the unique pointer to the array. In the function, `(*d)` is the array and `(*d)[i]` is the *i* th element of the array.

Function `readStream` inputs double precision values from the input stream. There are two possibilities—the number of values in the stream is less than or equal to the capcity of the array, or greater than the capacity of the array. If the number of values in the stream is less than or equal to the capacity of the array the function reads all the values into the array. If the number is greater than the capacity the function fills the array to its capacity and leaves the remaining values in the stream unprocessed. The statement that is responsible for deciding when to stop inputting is

```
for (int i = 0; i < cap && is >> (*d)[i]; i++)
```

It uses a common C++ pointer idiom. The input stream is the class reference `is`, which is a pointer. The expression

```
is >> (*d)[i]
```

attempts to input a value into `(*d)[i]`. If there is a value in `is` to input, `(*d)[i]` gets the value and the expression returns a non `nullptr` value. Because pointers are equivalent to integers, and nonzero integers are interpreted as true, the `for` loop continues. If there is no value to input, the expression returns `nullptr`, which is equivalent to 0 and interpreted as false. The loop terminates.

The two statements that are commented out in the main program

```
arr[2 * cap] = 123.4;
cout << arr[2 * cap] << endl;
```

are to illustrate what happens when you program with C++ primitive arrays. The first statement stores a value outside the boundary of the array, and the second statement outputs the value from the same location. Because C++ does not check its array bounds at execution time, the above statements will execute and even occasionally work. The first statement clobbers some cell in main memory in an unpredictable way. If the damage is benign the program will work. But if the corruption is fatal the program will crash. The following section constructs a safe array class that automatically checks for out-of-range indexing when you access an element of the array.

2.2 Array Classes

The objective of this section is to construct a class that behaves like an array but that does not permit the corruption that occurs if a value is stored outside the range of the index. An array is a collection of values, all of which have the same type. So, the question arises, What type should this safe array be? If you design the array to hold integers, your client is sure to want an array of doubles, and if you design an array of doubles, the client will want an array of strings.

One alternative is to use the `typedef` facility described in Section A.4 of the Appendix. At the beginning of the class implementation you could place the definition

```
typedef int T;
```

which makes the name `T` a synonym for `int`. Everywhere the implementation needs to refer to the type you write `T` instead of `int`. If the client needs a safe array of doubles, you replace the one line above with

```
typedef double T;
```

and recompile the implementation. The problem with this approach is that someone still needs to change a line of code with a text editor and recompile the app. If you want to store this class as a service in a software library it would not be feasible to require clients to modify the source code they want to use. You could make available many different compiled servers for many different types, but that would be wasteful and difficult to maintain. Furthermore, how could you anticipate all the possible types that a client might need?

Templates

The template facility of C++ is designed to solve the problem of programming a service when the type to be used by the client is not known. The advantage of using a template instead of a `typedef` is that you can write the implementation of a class for a generic type and provide just one copy of the implementation as a service. Two different clients can use the service with two different types, yet only one generic class need be provided in the library. Writing a service with a generic type using templates is referred to as *generic programming*. The beauty of generic programming in C++ is that there is no performance penalty compared to the direct approach. That is, client programs that use the services of a template class execute just as fast as if the class were provided with the desired type built in.

The concept of a template is similar to the concept of a function that provides a parameter list for its clients. For example, the greatest common divisor function

```
int gcd(int m, int n)
```

has formal parameters `m` and `n`. A client calls `gcd` with specific integer values as actual parameters for `m` and `n`. With a template class, the generic type is like a formal parameter. The client supplies a specific type as the actual parameter. Because passing a type to a template is similar to passing an actual parameter to a formal parameter, the template facility is also known as parametric polymorphism.


```
template <class T>
class ASeq {
public:
    explicit ASeq(int cap = 0) {} // Avoid implicit conversion.
    virtual ~ASeq() = default;
    virtual T &operator[](int i) = 0; // For read/write.
    virtual T const &operator[](int i) const = 0; // For read-only.
    virtual int cap() const = 0;

private:
    ASeq(ASeq const &rhs); // Disabled.
    ASeq &operator=(ASeq const &rhs); // Disabled.
};
```

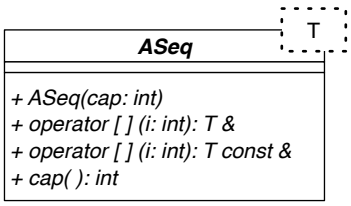


Figure 2.10 `ASeq.hpp`. The template class for an abstract sequence.

An abstract sequence class

Figure 2.10 shows the template class for the `ASeq` abstract class. An array is a specific example of a more general sequence. The mathematical notion of a sequence is a function f from the integers to a set of data of type T indexed by integers. For example, the sequence

40 20 70 50 10

is a set of data of type `int` indexed by integers 0..4 with the function f defined as

$$f(0) = 40, f(1) = 20, f(2) = 70, f(3) = 50, f(4) = 10$$

A specific implementation of an abstract sequence encloses the index in square brackets. For example, if `arr` inherits from `ASeq` and its elements are the integers in the above sequence, then `arr[2]` has the value 70.

The dashed box in the UML diagram corresponds to

```
template <class T>
```

in the code. It is like a formal parameter for the type. The actual parameter could be `int`, `double`, or some other type or class. Parameter `cap` in the constructor is the capacity of the sequence with a default value of 0. Any subclass of `ASeq` is required to implement the square bracket operator specified by

```
virtual T &operator[](int i) = 0;
virtual T const &operator[](int i) const = 0;
```

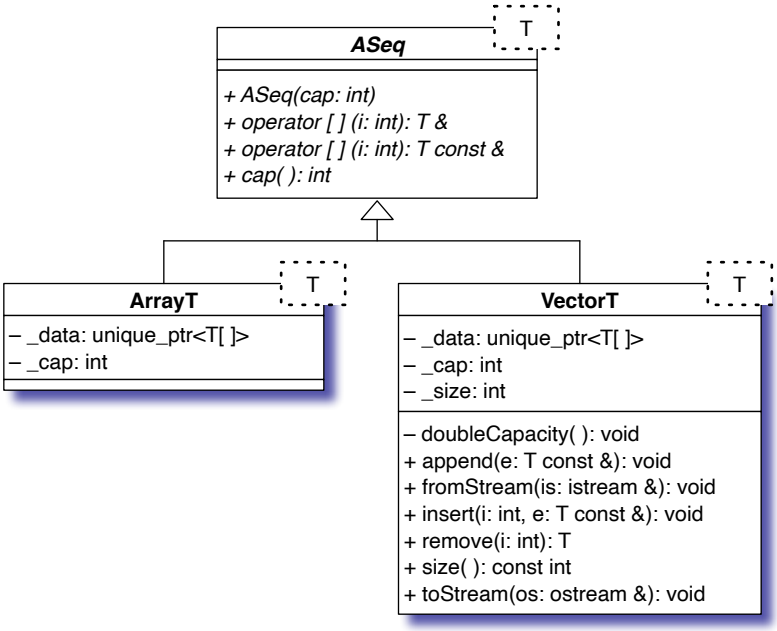



Figure 2.11 The UML diagram for the template classes that provide safe arrays and vectors.

It must also implement function `cap()`, which returns the capacity of the sequence.

Figure 2.11 is the UML diagram for `ArrayT` and `VectorT`, two specific classes in the dp4ds Distribution that inherit from the general `ASeq` abstract sequence class. `ArrayT` is a safe array that guards against accessing memory outside the range of the array. `VectorT` is a sequence that automatically expands during program execution to accommodate an increase in the amount of data being processed.

A safe array of doubles

Figure 2.12 is a main program that uses a template class that implements a safe array. `ArrayT.hpp` implements the class. Because `main()` calls no functions other than those in various libraries, there is no corresponding `ArrayMain.hpp` file. After reading the virtues of templates compared to `typedef`, you may wonder about the purpose of the statement

```
typedef ArrayT<double> ArrayDouble;
```

Its purpose is strictly a convenience. You could eliminate the above `typedef`, and everywhere in the program that `ArrayDouble` appears, simply replace it with `ArrayT <double>`. The safe array is provided as the `ArrayT` class. A template class requires the client to supply a type as an actual parameter. Rather than enclose the parameter list in parentheses as is the case with a function's parameter list, the list is enclosed in angle brackets `< >`. In the above statement, `double` is the actual parameter for the template class. The client is defining `ArrayDouble` to be a type that corresponds to the template class safe array for storing double precision real values.


```
#include <cstdlib> // EXIT_SUCCESS.
#include <iostream> // cout.
#include <fstream> // ifstream.
#include "Utilities.hpp"
#include "ArrayT.hpp"
using namespace std;

typedef ArrayT<double> ArrayDouble;

int main() {
    ArrayDouble arr(promptIntGE("Enter array capacity", 1));
    ifstream ifs;
    promptFileOpen(ifs);
    if (ifs) {
        int length = 0;
        readStream(ifs, arr, length);
        ifs.close();
        cout << "Read count == " << length << endl;
        cout << "Array data:" << endl;
        writeStream(cout, arr, length);
        // arr[2 * arr.cap()] = 123.4;
        // cout << arr[2 * arr.cap()] << endl;
        return EXIT_SUCCESS;
    }
}
```

Figure 2.12 `ArrayTMain.cpp`. A main program that uses a dynamically allocated safe array of doubles from a template class.

The first statement in the main program

```
ArrayDouble arr(promptIntGE("Enter array capacity", 1))
```

makes object `arr` an instantiation of the class `ArrayDouble`. The class provides a constructor with a parameter list having a single integer parameter, which is executed when `arr` is declared. Whatever integer value the user enters is used internally in the class to set the capacity of the dynamically allocated array.

The main program is short because functions `readStream()` and `writeStream()` are provided along with the template class in the library. They perform input and output like the corresponding functions in Figures 2.8 and 2.9 for the primitive array. There is no need to supply the capacity of the array as a parameter to these versions of the functions because the array class stores the capacity of the array as an attribute.

The statements that are commented out

```
arr[2 * arr.cap()] = 123.4;
cout << arr[2 * arr.cap()] << endl;
```



```
// ========= ArrayT =========
template<class T>
class ArrayT : public ASeq<T> {
private:
    unique_ptr<T[]> _data;
    int _cap;
public:
    explicit ArrayT(int cap = 1);
    int cap() const override;
    T &operator[](int i) override; // For read/write.
    T const &operator[](int i) const override; // For read-only.
private:
    ArrayT(ArrayT const &rhs); // Disabled.
    ArrayT &operator=(ArrayT const &rhs); // Disabled.
};

// ========= Constructor =========
template<class T>
ArrayT<T>::ArrayT(int cap) {
    if (cap < 1) {
        cerr << "ArrayT constructor precondition 0 < cap violated." << endl;
        cerr << "cap == " << cap << endl;
        throw -1;
    }
    _data = make_unique<T[]>(cap);
    _cap = cap;
}

// ========= cap =========
template<class T>
int ArrayT<T>::cap() const {
    return _cap;
}
```

Figure 2.13 `ArrayT.hpp`. The template class that provides a safe array. The listing continues in the next figure.

show that you can access the elements of `arr` with square brackets `[]` as if it were a primitive array. Unlike a primitive array, however, execution of the above assignment statement is guaranteed to not corrupt your computer in some unknown way. Instead, a precondition will be violated, an error message will direct you to the offending code, and the program will terminate.

Figure 2.13 is the listing of the template class for the safe array. It turns out to be difficult with C++ to separate the specification from the implementation of a template service. Consequently, the template class `ArrayT` is both specified and implemented in the `.hpp` file. There is no corresponding `.cpp` file.


```
// ========= operator[] =========
template<class T>
T &ArrayT<T>::operator[](int i) {
    if (i < 0 || _cap <= i) {
        cerr << "ArrayT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

template<class T>
T const &ArrayT<T>::operator[](int i) const {
    if (i < 0 || _cap <= i) {
        cerr << "ArrayT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}
```

Figure 2.14 `ArrayT.hpp` (continued). Overloading the subscript operator `[]`. The listing continues in the next figure.

Attribute `_data` is declared as

```
unique_ptr<T[]> _data;
```

It is a pointer to an array of elements of type `T`. Allocation in the constructor is accomplished with

```
_data = make_unique<T[]>(cap);
```

which allocates the array from the heap. Again, because `T` is the formal parameter of the template, if the actual parameter in the client were `double`, the allocation in the constructor would have the same effect as

```
_data = make_unique<double[]>(cap);
```

The constructor sets attribute `_cap` to the value passed to it as a parameter.

Overloading operator [ ]

Figure 2.14 shows the implementation of the methods that allow `main()` to access elements of the class by indexing as if the class were a primitive array. The idea is to treat the subscript brackets `[]` as an operator, and overload the operator name. When a client writes an object name followed by the subscript brackets, C++ will detect the class of the object and determine if the subscript operator has been defined for that class. If it has, the compiler will invoke the method.

There are two ways to declare an array—with `const` and without. In the same way that the value of a constant integer cannot be changed, the values of a constant array


```
// ========= readStream =========
template<class T>
void readStream(istream &is, ArrayT<T> &a, int &len) {
    // Pre: >> is defined for T.
    // Post: num values are input from is to a, where
    // num == min(number of elements in is, a.cap()).
    // len == num.
    len = 0;
    int cap = a.cap();
    for (int i = 0; i < cap && is >> a[i]; i++) {
        len++;
    }
}

// ========= writeStream =========
template<class T>
void writeStream(ostream &os, ArrayT<T> const &a, int len) {
    // Pre: << is defined for T.
    // Post: num values are output from a to os, six items per line,
    // where num == min(len, a.cap()).
    int cap = a.cap();
    for (int i = 0; i < len && i < cap; i++) {
        os.width(16);
        os.precision(6);
        os.setf(ios::fixed | ios::showpoint);
        os << a[i];
        if (i % 6 == 5) {
            os << endl;
        }
    }
    os << endl;
}
```

Figure 2.15 `ArrayT.hpp` (continued). The `readStream` and `writeStream` functions, which are called from the program in Figure 2.12. This completes the listing.

cannot be changed either. It is possible to initialize the values of a constant array when the array is first declared. Once initialized, the values will never change. Another use of a constant array is as a formal parameter in a function's parameter list. Even if the actual parameter is not a constant array, the compiler will not permit any changes to be made to the elements of the formal parameter within the scope of the function.

Corresponding to these two ways to declare an array, class `ArrayT` must implement two versions of the overloaded index operator. For an object that does not have `const`, it implements the method


```
T &ArrayT<T>::operator[](int i)
```

You can think of `operator[]` as the name of the function and `(int i)` as its formal parameter list. With the array declared as in Figure 2.12, you might write a statement like

```
num = arr[13];
```

C++ treats the index 13 as the actual parameter, which corresponds to formal parameter `i`. The function returns `T &`, which is a reference to `T`. Because the actual type is `double` in Figure 2.12, you can think of the function as returning `double &`, that is, a reference to a double precision real.

The first statement in the method

```
if (i < 0 || _cap <= i) {
   cerr << "ArrayT index out of bounds: "
        << "index == " << i << endl;
   throw -1;
}
```

is what makes the array safe. The `if` test will be true if the value supplied by the actual parameter is outside the allowable range. In the above example, if the capacity of the array is 8, and the array is subscripted with 13, the precondition is not met, and the `if` statement executes. An appropriate error message is generated and your application will not be corrupted in some unknown way.

The second statement in the method

```
return _data[i];
```

uses the index supplied by the client to access the primitive array stored as an attribute in the class. Because the type returned is `T &`, the `return` statement returns a reference to `_data[i]`. For example, suppose the main program of Figure 2.12 executes the statement

```
arr[13] = 7.1;
```

The assignment operator = expects a reference to a memory cell on its left hand side to which it will assign a value. The return statement obliges by returning a reference to `_data[13]`, which then gets the value 7.1.

The second version of the overloaded operator is for arrays that are declared with `const`. It is identical to the first version except for its heading

```
T const &ArrayT<T>::operator [](int i) const
```

And in the heading the only difference is the existence of the keyword `const` in two places. The first `const` applies to the return type `T const &`. This `const` informs the compiler that the values of the constant array cannot change. The second `const` makes the method a constant member function. A constant member function is prevented from modifying any attributes of its class.

For example, suppose `arr` is a constant array in the formal parameter list of a function as follows.

```
void Alpha(ArrayDouble const &arr)
```

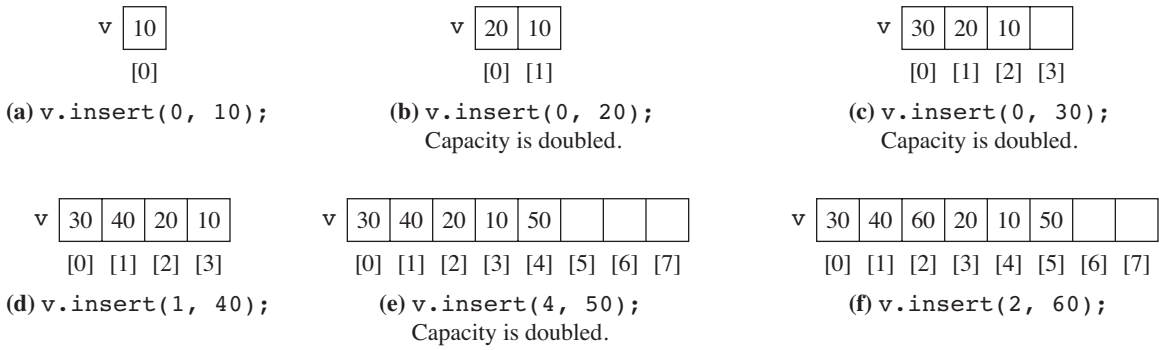



Figure 2.16 The effect of the insert operation on vector `v`. The first parameter of the insert method is the index of where to insert the value. The second parameter is the value to be inserted.

If the compiler encounters the following statement within the function

```
num = arr[13];
```

it notes that `arr` is a constant class that uses the index operator. So, it looks for a constant member function of that class that implements the overloaded index operator and finds this second version. It notes that the return type is `const T&`, which cannot change. No problem here, because the returned value is used on the right hand side of the assignment.

On the other hand, if the compiler encounters

```
arr[13] = 7.1;
```

it notes again that `arr` is a constant class, looks for a constant member function, and finds the second version. But now it notes that the return type is `T const &`, which does not permit the referenced cell, `a[13]`, to change. Therefore, the compiler issues an error and does not permit the assignment.

2.3 A Vector Class

One annoying feature about arrays is that you must specify how many cells will be in the array when you allocate it. This is true whether it is allocated on the stack and you must commit to its capacity at compile time, or whether you allocate it dynamically during program execution. In both cases, you are committed to the capacity of the array. After you make that commitment and begin populating the array with values, you can no longer increase the capacity of the array.

Properties of vectors

A vector is a data structure that is similar to an array because you access its values with the usual square bracket operator. For example, if `v` is a vector of `int`, and you want to set its third element to 50, you execute


```
template<class T>
class VectorT : public ASeq<T> {
private:
    unique_ptr<T[]> _data;
    int _cap;  // Invariant: 0 < _cap, and _cap is a power of 2.
    int _size; // Invariant: 0 <= _size <= _cap.

    void doubleCapacity();

    VectorT(VectorT const &rhs); // Disabled.
    VectorT &operator=(VectorT const &rhs); // Disabled.
```

Figure 2.17 Specification of the private part of the vector class `VectorT`.

```
v[2] = 50;
```

In addition to this usual random access feature, a vector provide two advantages over an array:

- Its capacity increases automatically even after you begin populating it with values.
- It provides an insert operation that shifts current values to the right to accommodate the inserted value, and a remove operation that shifts current values to the left to fill the cell whose value is removed.

The capacity of a vector begins at one and automatically doubles when necessary to accommodate a new value. Figure 2.16 shows both these features for vector `v`. At a given point in time a vector has a size and a capacity, with the invariant that the size is always less than or equal to the capacity. In Figure 2.16(d), the size and the capacity are both four. In Figure 2.16(e), to insert the value 50 at index four the size increases to five and the capacity doubles to eight.

A vector implementation

Vector class `VectorT` inherits from the abstract sequence `ASeq` in the same way that `ArrayT` does. Figure 2.11 shows the UML diagram for `VectorT` and Figure 2.17 is the specification if its private part. A vector has three attributes. `_data` is the raw array of values, `_cap` is the capacity of the vector, and `_size` is its size.

Figure 2.18 shows the implementation of `doubleCapacity()`, which doubles the capacity of the vector. First, it allocates a new array named `newDat` with twice the capacity of the current array attribute `_data`.

```
_cap *= 2;
T *newDat = new T[_cap];
```

`newDat` is a pointer to a raw array, not a `unique_pointer` to a raw array. It is allocated with the `new` operator, and, by itself, there would be no automatic garbage collection. Second, it copies the values from `_data` to `newDat` in a `for` loop. Third, the statement


```
// ========= doubleCapacity =========
template<class T>
void VectorT<T>::doubleCapacity() {
    _cap *= 2;
    T *newDat = new T[_cap];
    for (int k = 0; k < _size; k++) {
        newDat[k] = _data[k];
    }
    _data.reset(newDat);
}
```

Figure 2.18 Implementaation of the `doubleCapacity()` method of the vector class `VectorT`.

```
_data.reset(newDat);
```

makes `newDat` an attribute of the `_data` object, which is a `unique_pointer` to a raw array. Automatic garbage collection does deallocate the old `_data` array.

An alternative strategy would be to increase the capacity by one, which would conserve memory. However, such a strategy would be more time consuming because you would have to copy over the entire array each time an element is appended. With the doubling strategy, you anticipate that additional append operations will execute in the future and preallocate storage for them.

Figure 2.19(a) shows the specification of the constructor and part (b) shows its implementation. The constructor allocates a new array of type `T` with one cell and sets `_cap` to one and `_size` to zero. The declarations and implementations of `cap()` and `size()` in Figure 2.19(a) are combined because they are simple one-liners.

Figure 2.20 shows that method `append()` has no precondition. You can append a value to an empty vector. And even if you append a value to a vector that is full, it will automatically double its capacity to accommodate the appended value.

Compare the preconditions of `insert()` and `remove()`. The precondition for `insert()` is

```
// Pre: 0 <= i && i <= size().
```

and the precondition for `remove()` is

```
// Pre: 0 <= i && i < size().
```

The preconditions differ because you can insert an element after the last one in the vector, but you cannot remove an element after the last one. That is, if `i` has the value `size()` then the precondition for `insert()` is satisfied, but the precondition for `remove()` is not. Furthermore, you can insert an element in an empty vector, but you cannot remove an element from one. That is, if the value of `size()` is zero and `i` is also zero the precondition for `insert()` is satisfied, but the precondition for remove is not because $0 < 0$ is false.

Figure 2.21 shows that overloading the `[]` operator is accomplished as it is with the safe array classes.


```
public:
    VectorT();
    // Post: This vector is initialized with capacity of 1 and size of 0.

    int cap() const override { return _cap; }
    // Post: The capacity of this vector is returned.

    int size() const { return _size; }
    // Post: The size of this vector is returned.
```

(a) Specification of the constructor. Specification and implementation of `cap()` and `size()`.

```
// ========= Constructor =========
template<class T>
VectorT<T>::VectorT() {
    _data = make_unique<T[]>(1);
    _cap = 1;
    _size = 0;
}
```

(b) Implementation of the constructor.

Figure 2.19 The constructor, `cap()`, and `size()` methods of the vector class `vectorT`.

The safe array `ArrayT` provides `readStream()` to input values from an input stream and `writeStream()` to output values to an output stream. Figure 2.15 shows that these functions are not methods, i.e. member functions. They take an input or output stream, an `ArrayT`, and an integer length as parameters. `VectorT` handles input/output differently. Figure 2.22 shows that function `toStream()` is a method and has only one parameter, an output stream. Because `VectorT` has attribute `_size` there is no need for a length parameter.

Figure 2.22 shows how to overload `operator<<` so vectors can use the binary `<<` output operator. The overloaded `operator<<` cannot be a method of `VectorT`, because of the requirements that C++ places on its signature. It must return a reference to an input stream, its first parameter must be an input stream, and its second parameter must correspond to the right hand side (rhs) of the `<<` operator. In this case, `rhs` is a `VectorT`.

You might be tempted to dispense with method `toStream()` altogether and incorporate its logic into `operator<<`. The problem with that approach is that `toStream()` needs access to the private attributes of `VectorT` to be able to output a representation of the vector. But `operator<<` is not a method, and therefore does not have access to the attributes of the vector. Because `toStream()` is a method it does have access, and so can use the attributes to generate the stream of characters to the output stream.

Figures 2.23 and 2.24 show the listing of a main program to test the implementa-


```
public:
    void append(T const &e);
    // Post: Element e is appended to this vector, possibly increasing cap().

    void insert(int i, T const &e);
    // Pre: 0 <= i && i <= size().
    // Post: Items [i..size()-1] are shifted right and element e is
    // inserted at position i.
    // size() is increased by 1, possibly increasing cap().

    T remove(int i);
    // Pre: 0 <= i && i < size(). T has a copy constructor.
    // Post: Element e is removed from position i and returned.
    // Items [i+1..size()-1] are shifted left.
    // size() is decreased by 1 (and cap() is unchanged).
```

(a) Specification of `append()`, `insert()`, and `remove()`.

```
// ========= append =========
template<class T>
void VectorT<T>::append(T const &e) {
    if (_size == _cap) {
        doubleCapacity();
    }
    _data[_size++] = e;
}
```

(b) Implementation of `append()`.

Figure 2.20 Methods to append to, insert into, and remove from `VectorT`. Implementation of `insert()` and `remove()` are exercises for the student.

tion of the `VectorT` data structure. It prompts the user for a one-letter response, then depending on the response, calls one of the `VectorT` methods.

Unit tests

After you implement a data structure you must test it to make sure it satisfies its specification. Most data structures have many methods, which raises the question of how to test the data structure. One approach would be to write all the methods of the data structure, then write some application that uses the data structure, then test the application to see how well it works. The problem with this approach is that if you discover a bug in the application you do not know if the error is in the application or in the data structure.

To alleviate this problem, common software engineering practice is to provide what is known as a *unit test* to test an individual method of a data structure. The idea is to thor-


```
T &operator[](int i) override; // For read/write.
T const &operator[](int i) const override; // For read-only.
```

(a) Specification of `operator[]`.

```
// ========= operator[] =========
template<class T>
T &VectorT<T>::operator[](int i) {
    if (i < 0 || _size <= i) {
        cerr << "VectorT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

template<class T>
T const &VectorT<T>::operator[](int i) const {
    if (i < 0 || _size <= i) {
        cerr << "VectorT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}
```

(b) Implementation of `operator[]`.

Figure 2.21 Overloading the `[]` operator of the vector class `VectorT`.

oughly test each individual method of a data structure in isolation from the other methods and from any application that will use the data structure. Then, when you encounter a bug in an application you can rule out any errors in the data structure implementation, which in turn makes it easier to track down the bug.

Many systems for unit testing exist, each one depending on the programming language and the integrated development environment (IDE) that the programmer uses. Instead of using a commercial unit test system, all the data structures projects in the dp4ds Distribution for this book use a unit test system based on the main program for the project as a driver.

Figure 2.25 shows the unit test for the insert method of `VectorT`. The first group of lines in the file represent the user input for the main program in Figures 2.23 and 2.24. For example, the first line in Figure 2.25 is

```
i 10 0 w c s
```

You can see from Figure 2.23 that if you run the main program it will prompt you for a one-letter response. If you enter `i` as in the first line above, the program will branch to case `'I'` then prompt you for the value to insert. If you enter `10` for the value, it will prompt you for the location to insert. If you enter `0`, it will prompt you for another one-


```
// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, VectorT<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void VectorT<T>::toStream(ostream &os) const {
    os << "(";
    for (int i = 0; i < _size - 1; i++) {
        os << _data[i] << ", ";
    }
    if (_size > 0) {
        os << _data[_size-1];
    }
    os << ")";
}

// ========= operator>> =========
template<class T>
istream &operator>>(istream &is, VectorT<T> &rhs) {
    rhs.fromStream(is);
    return is;
}

// ========= fromStream =========
template<class T>
void VectorT<T>::fromStream(istream &is) {
    T temp;
    while (is >> temp) {
        append(temp);
    }
}
```

Figure 2.22 The input/output streaming operators of the vector class `VectorT.hpp`.

letter response. If you enter `w`, it will write the data structure to the output stream and prompt you for another one-letter response. Entering `c` then `s` will cause the capacity and size to output. If the input method is implemented correctly, the result of inputting the above line will produce the output

```
(10)
The capacity is 1
```



```
typedef VectorT<int> VectorInt;

int main() {
    VectorInt v;
    int value, index;
    ifstream ifs;
    char response;
    do {
        cout << "\n(c)ap  (s)ize  (a)ppend  (f)ileAppend  (i)nsert  "
             << "(r)emove  se(t)  (w)rite  (q)uit: ";
        cin >> response;
        switch (toupper(response)) {
        case 'C':
            cout << "\nThe capacity is " << v.cap() << endl;
            break;
        case 'S':
            cout << "\nThe size is " << v.size() << endl;
            break;
        case 'A':
            cout << "Append what integer value? ";
            cin >> value;
            v.append(value);
            break;
        case 'F':
            promptFileOpen(ifs);
            if (ifs) {
                ifs >> v;
                ifs.close();
            }
            break;
        case 'I':
            cout << "Insert what integer value? ";
            cin >> value;
            cout << "Insert at what location? ";
            cin >> index;
            v.insert(index, value);
            break;
        case 'R':
            cout << "Remove from what location? ";
            cin >> index;
            value = v.remove(index);
            cout << "\n" << value << " removed." << endl;
            break;
```

Figure 2.23 `VectorTMain.cpp`. A main program to test `VectorT`. The listing continues in the next figure.


```
        case 'T':
            cout << "Set what integer value? ";
            cin >> value;
            cout << "Set at what location? ";
            cin >> index;
            v[index] = value;
            cout << "\nValue at index " << index << " is now "
                 << v[index] << endl;
            break;
        case 'W':
            cout << "\n" << v << endl;
            break;
        case 'Q':
            break;
        default:
            cout << "\nIllegal command." << endl;
            break;
        }
    } while (toupper(response) != 'Q');
    return EXIT_SUCCESS;
}
```

Figure 2.24 `VectorTMain.cpp` (continued). A main program to test `VectorT`. This completes the listing.

```
The size is 1
```

as shown in Figure 2.25.

Similarly, entering the other lines at the top of Figure 2.25 will produce the output shown in the rest of the figure if method `insert()` is implemented correctly. So, to test your implementation you would need to enter the sequence of prompts shown at the top of the figure and compare them with the expected output shown at the bottom of the figure. Note that the last one-letter response is `q` which terminates the main program.

Fortunately, you do not need to manually enter the responses to run the unit test. Instead, you can redirect the standard input for the program to come from the unit test file instead of from the keyboard. Because the sequence of responses is at the top of the file and the last `q` will terminate the main program, the program will not encounter the remainder of the file in its input stream.

The above technique is convenient if you are developing in a command line environment. If you are running in an IDE it should have a way to redirect the input to come from a file. However, a more convenient way to run a unit test in an IDE is to simply run the main program and wait for its first prompt in the console pane. Then you can simply copy the responses from the top of the unit test file and paste them into the console pane. Most IDEs will take the paste as if the stream of characters are entered from the keyboard. This technique usually works in a command line environment as well.

Every project in the dp4ds Distribution comes with a set of unit tests and a main program to drive them. For brevity, none of the later chapters show the main program


```
i 10 0 w c s
i 20 0 w c s
i 30 0 w c s
i 40 1 w c s
i 50 4 w c s
i 60 2 w c s q

VectorT unit-insert

(10)
The capacity is 1
The size is 1

(20, 10)
The capacity is 2
The size is 2

(30, 20, 10)
The capacity is 4
The size is 3

(30, 40, 20, 10)
The capacity is 4
The size is 4

(30, 40, 20, 10, 50)
The capacity is 8
The size is 5

(30, 40, 60, 20, 10, 50)
The capacity is 8
The size is 6
```

Figure 2.25 `unit-insert.txt`. The unit test for the insert method of `VectorT`. This unit test is contained in the dp4ds distribution software for this book. The sequence of insertions corresponds to those of Figure 2.16.

driver or the unit tests. However, you should avail yourself of the unit tests when asked to implement a method of a data structure.

Exercises

2–1 Execute the main program `ArrayClassicMain` from Figure 2.9. Verify that it works correctly when the number of values in the input stream is less than the capacity of the array and when it is greater. Then remove the comment characters `//` to execute the

statements that access memory beyond the range of the array. Experiment to find a value for the capacity of the array that will allow the out-of-bounds reference to apparently work correctly. Experiment to find a value that will crash your program.

2–2 Execute the main program `ArrayTMain` from Figure 2.12. Verify that it works correctly when the number of values in the input stream is less than the capacity of the array and when it is greater. Then remove the comment characters `//` to execute the statements that access memory beyond the range of the array. What error message do you get?

2–3 Implement the methods `insert()` and `remove()` in `VectorT.hpp`. Be sure to implement the preconditions. Test your implementation with the unit tests in the dp4ds Distribution software for those cases that satisfy the preconditions. Test with interactive input to verify that your preconditions are implemented correctly. For example, if you enter `-1` for the value of the index in remove the error message should be

```
VectorT remove precondition 0 <= i && i < size() violated.
i == -1
```

2–4 The specification of `VectorT` never calls for decreasing the capacity. Modify the implementation of Exercise 2–3 so that when the size decreases to one fourth the capacity, the capacity decreases by one half. Maintain the invariant on `_cap` specified in Figure 2.17. Devise a new unit test named unit-collapse to test your feature. Use the format of Figure 2.25 including the input stream of responses to the main program and the listing of the expected output for a successful test.

2–5 The text (page 52) gives the output of Figure 2.7 if you delete the `&` symbols in the parameter list. Draw the memory allocation corresponding to Figures 2.7(a) and (b) for this case immediately before the return from the function.

