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?

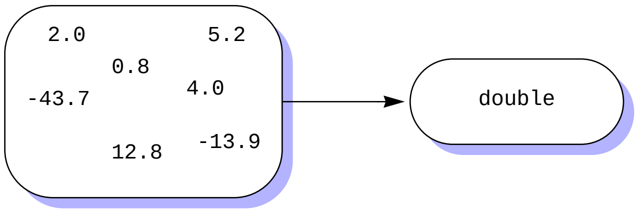


Figure 1.1 Type abstraction for type `double`. In the C++ programming language, an expression of type `double` must have as its value one of many possible specific values that define the type. The same principle holds for other types. For example, the values `true` and `false` define type `bool`.

> I do.
>
> 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.

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. 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.

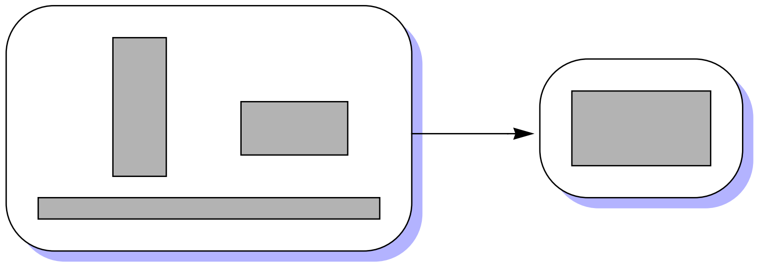


Figure 1.2 Structure abstraction to abstract from specific shapes of many different sizes to a single shape with a general size.

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 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

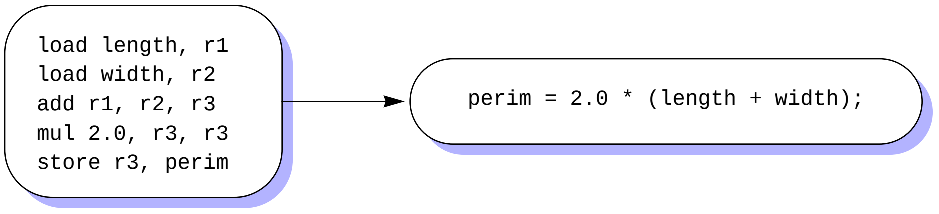


Figure 1.3 Statement abstraction for the assignment statement. In C++, the assignment operator evaluates the expression on its right hand side and gives the value to the variable on the left hand side.

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.

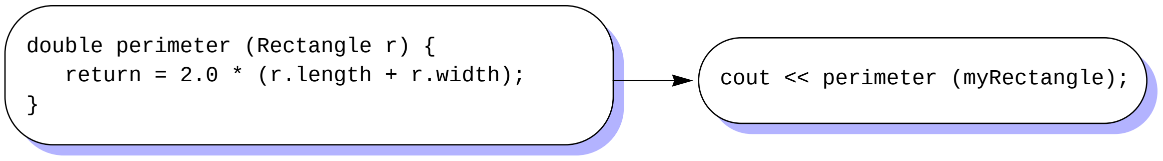
Corresponding to structure abstraction on the data side of the coin is procedure abstraction on the computation side. In the same way that high-level languages allow you to collect variables into structures to create a new data type, they allow you collect statements into procedures to create a new computation.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



```
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 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

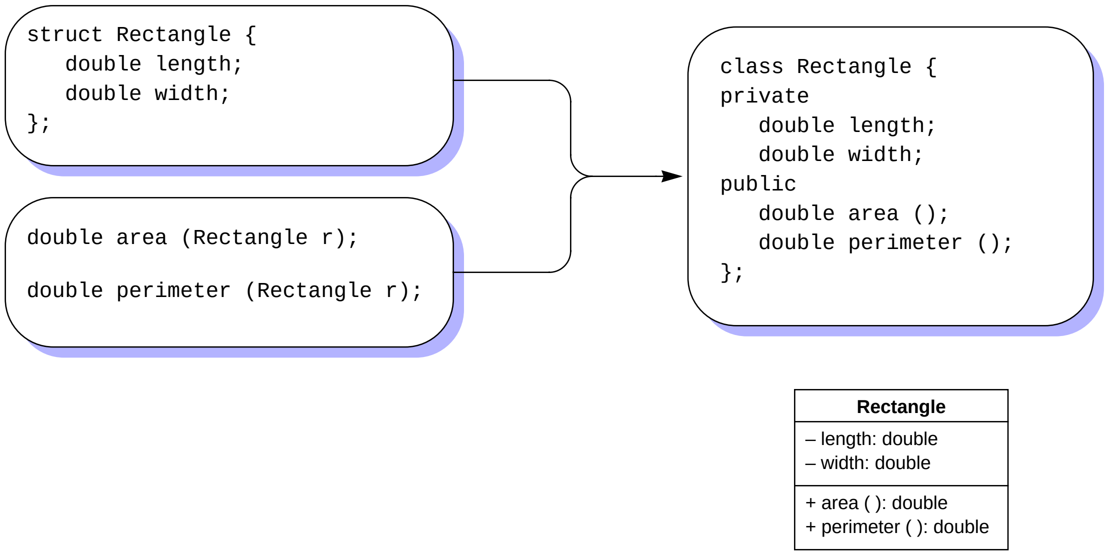
```
temp = gcd(num, denom);
```

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.



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

```
double area (Rectangle r);

double perimeter (Rectangle r);
```

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

Figure 1.5 Class abstraction that combines the structure abstraction of Figure 1.2 with the procedure abstraction of Figure 1.4. Data and computation are combined in the definition of the class `Rectangle`. The box in the lower part of the figure shows a Unified Modeling Language (UML) depiction of the class.

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

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.

To emphasize the shift in focus when a function is bound to a type, object-oriented

UML	C++
class	class
object	object
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++.

designers established a new set of terminology. Roughly speaking, in object-oriented terminology

- *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 1.2(b), 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

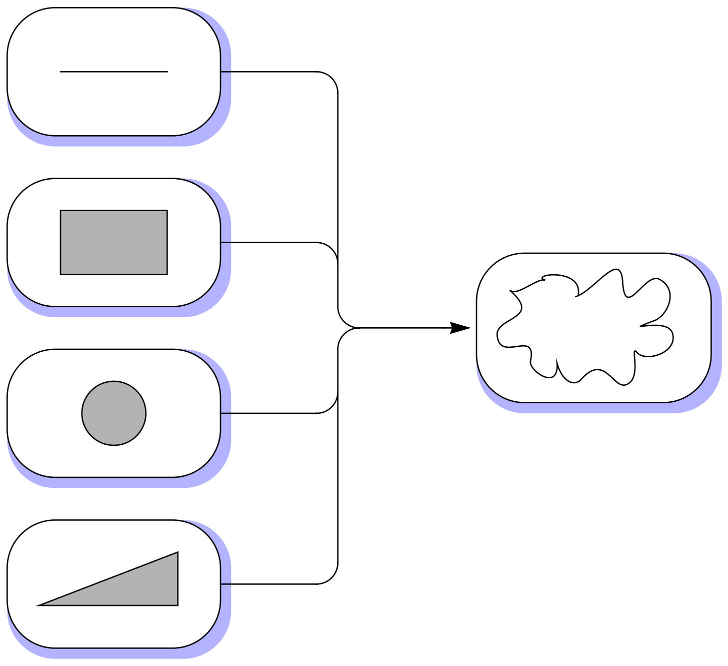
```
double Rectangle::perimeter () {
   return 2.0 * (length + width);
}
```

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?

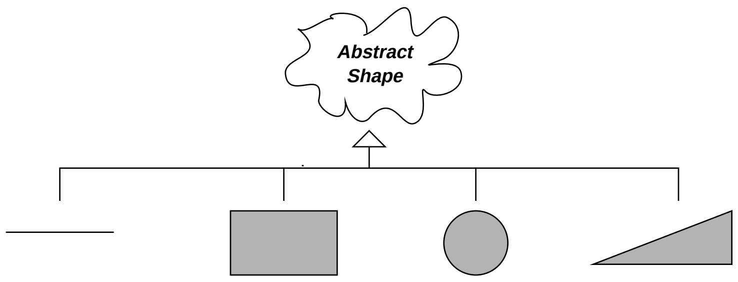
The answer is that there is, in effect, an implicit formal parameter not shown in the definition of the method. The corresponding actual parameter is, however, explicit when the method is called. To output the perimeter of object `myRectangle`, you write

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

Compare this method call with the corresponding 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.



(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.

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.

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 `double`s—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 `double`s 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 declaration

```
class AShape : public virtual AObject
```

names the class. The `A` in `AShape` stands for abstract. The code in Figure 1.8 contains no objects. It simply declares the abstract class `AShape`. A colon following a class name is the C++ notation for inheritance. This declaration states that `AShape` inherits from `AObject`. Appendix A gives the declaration of `AObject` and a brief description of its purpose. `AObject` is required for the classes in this book because C++, unlike many object-oriented languages, does not provide a universal base class from which all

```
// File: Ch01/Shape/AShape.hpp

#ifndef AShape_hpp
#define AShape_hpp

#include "AObject.hpp"

class ostream;  // Forward declaration.

class AShape : public virtual AObject {
   public:
      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 (ostream& os) = 0;
      // Post: This shape's name and dimensions are printed to os.

      virtual void promptAndSetDimensions () = 0;
      // Post: This shape's dimensions are prompted and set.
};

#endif
```

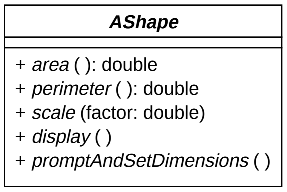


Figure 1.8 The content of the C++ header file for declaring the abstract shape class of Figure 1.7. The name of the class in the C++ code is AShape. The UML standard notation for an abstract class is to render the name of an abstract class in bold slanted type and the name of its methods in slanted type. 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.

other classes inherit. Some classes in later chapters will require that classes in earlier chapters inherit from a common base class. Rather than try to anticipate which classes will need to inherit from the universal base class, this book will simply design all classes to inherit from `AObject` unless they inherit directly from some other class. Inheritance from `AObject` will be assumed from now on, and will not be indicated in the UML diagrams.

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 can 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.

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 π 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.

The UML diagram in Figure 1.8 corresponds to the C++ code in the header file. A rectangular block in a UML diagram corresponds to a class. The topmost cell in the

```
// File: Ch01/Shape/Rectangle.hpp

#ifndef Rectangle_hpp
#define Rectangle_hpp

#include "AShape.hpp"

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

   public:
      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 ();
      double perimeter ();
      void scale (double factor);
      void display (ostream& os);
      void promptAndSetDimensions ();
};

#endif
```

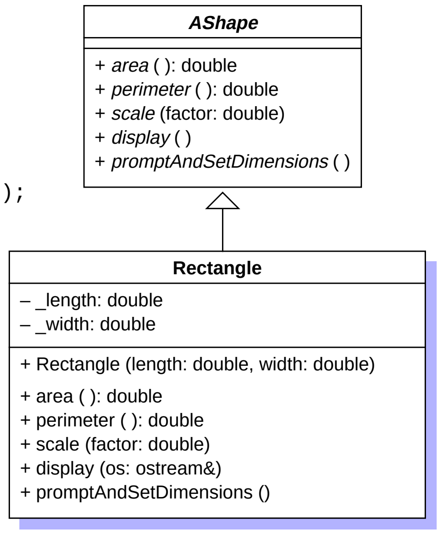


Figure 1.9 Specification for the concrete `Rectangle` class. The UML syntax for a private item is to precede it with the – symbol, and for a public item is to precede it with the + symbol. The block for the rectangle is divided into three parts—the name of the class, the attributes, and the operations. Our C++ style convention is to always begin the name of an attribute with the underscore character _.

block contains the name of the class. The figure shows that the syntax of C++ differs slightly from the syntax of a UML diagram. You should familiarize yourself with UML diagrams as they are a powerful tool for communication about object-oriented design and have become an industry standard. In particular, you should be able to make the connection between C++ program code and UML diagrams.

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;
```

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 he 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);
// 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` operator. 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 in Section A.7 in the appendix.

```
// File: Ch01/Shape/Rectangle.cpp

#include <iostream.h>
#include <assert.h>
#include <string.h>
#include "Rectangle.hpp"
#include "Utilities.hpp"

Rectangle::Rectangle (double length, double width) {
   assert (length >= 0.0 && width >= 0.0);
   _length = length;
   _width  = width;
}

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

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

void Rectangle::scale (double factor) {
   // Exercise for the student.
}

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

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

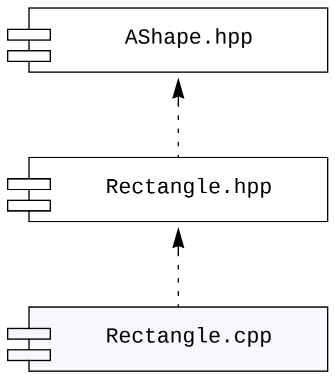


Figure 1.10 An implementation of the `Rectangle` class. The code for method `scale` in this and all the other shapes is left as an exercise for the student. Also shown is a UML component diagram that displays the chain of compile time dependencies beginning with this file `Rectangle.cpp`. The file that contains an implementation is shaded.

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 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.

A concrete class implementation

Figure 1.10 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 1.10 also has a component diagram that shows the compile time dependencies between the files in Figures 1.8, 1.9, and 1.10. The arrow from `Rectangle.cpp` to `Rectangle.hpp` means that `Rectangle.cpp` depends on `Rectangle.hpp` by virtue of the include statement in Figure 1.10.

The first line of method `area`,

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

begins with 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`

```
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 the `assert` function. For example, the precondition for the `Rectangle` constructor from Figure 1.9 is

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

which is implemented with the assert function call as

```
assert (length >= 0.0 && width >= 0.0);
```

in Figure 1.10. The effect of the assert function 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 which assert statement failed. 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

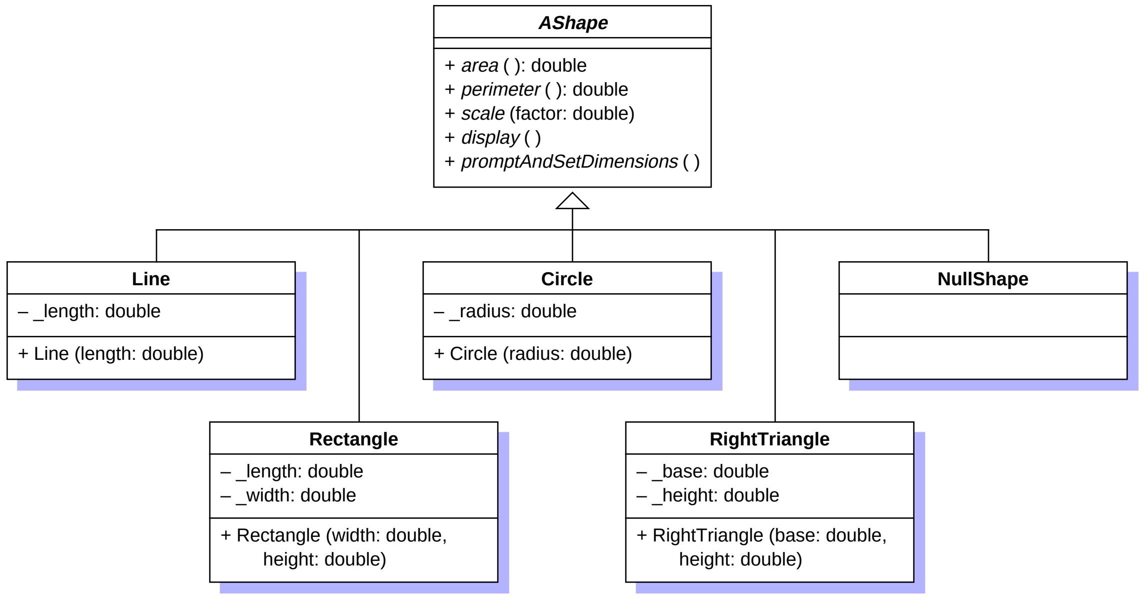


Figure 1.11 A class diagram for the abstract class `AShape` and the concrete classes `Line`, `Rectangle`, `Circle`, `RightTriangle`, and `NullShape`. Because each concrete class inherits and implements all the methods of the abstract class, they are not repeated in the operations part of the class box as they are in the class box of `Rectangle` in Figure 1.9. Shading is behind those classes that provide an implementation.

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 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`.

Similar implementation code, not shown here, is necessary for each concrete shape. Figure 1.11 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

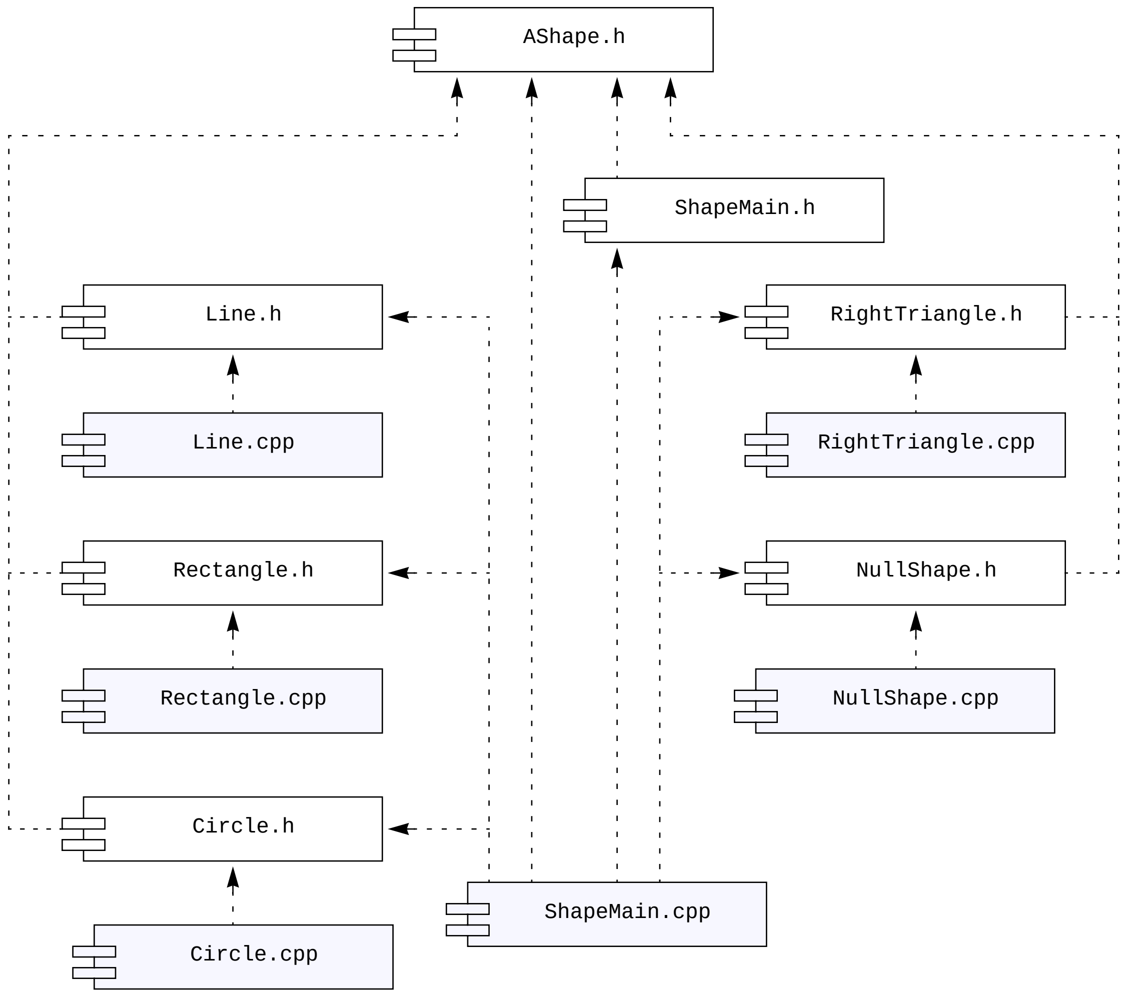


Figure 1.12 A UML component diagram of the compiler dependencies for the client program in `ShapeMain.cpp` that uses the shape classes. Some library files are not shown, such as the `assert.h` header file for the assert statement. Also omitted is a utility file `Utilities.h` that provides the constant π for the circle implementation and some routines to prompt the user for numeric input within specified limits.

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 what is known as the null object pattern.

A client application

Figure 1.12 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

```
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 object? 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 1.13 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 or dimensions of the shape. User responses are bold.

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 1.13 shows a sample interactive session produced by the main program.

Figure 1.14 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`.

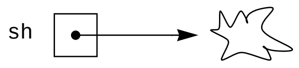
Each function is documented with a postcondition and possibly a precondition. Many of the functions have a side effect of prompting the user for some input, which is documented as well. It should be straightforward to compare the functions in Figure 1.14 with the interactive session in Figure 1.13 to determine which function is responsible for which prompt. For example, `shapeType` produces the prompt

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

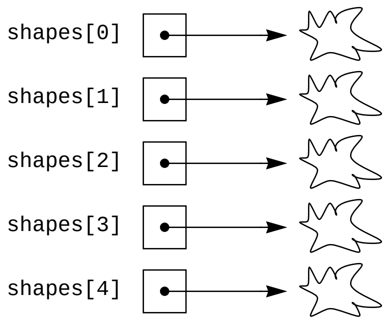
as a side effect. Its purpose is to return one of the uppercase letters L, R, C, T, or M. The user has no concept of the null shape, which is not one of the options. In addition to the line, rectangle, circle, and right triangle there is an option for a mystery shape. This

```
// File: Ch01/Shape/ShapeMain.hpp

#ifndef ShapeMain_hpp
#define ShapeMain_hpp

#include "AShape.hpp"

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

void cleanUp (AShape* shapes[], int cap);
// Pre: shapes[0..cap - 1] is allocated, and all elements are well-defined.
// Post: All shapes[0..cap - 1] are deleted and set to NULL.

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

void makeShape (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 (AShape*& sh);
// Post: Original sh is deleted and sh is made the null shape.

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

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

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

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

#endif
```

Figure 1.14 The header file for a main program that uses the shape classes. In Figure 1.12, 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 1.15 Visual depictions of the parameters of the functions in Figure 1.14. Each cloud represents one abstract shape as in Figure 1.7.

option is provided for the student to complete as an exercise.

The functions in Figure 1.14 have three different types of formal parameters. The simplest parameter is `AShape*` as in the function

```
void printArea (AShape* sh);
```

The asterisk symbol `*` is the C++ notation for a pointer. 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 1.15(a).

The second type of parameter is `AShape*&` as in the function

```
void makeShape (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 1.15(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 1.13, the user requested a rectangle, and so was prompted for the dimensions 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 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 for further discussion of the difference between call by reference and call by value.

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

```
void initialize (AShape* shapes[], int cap);
```

The brackets indicate that the formal parameter `shapes` is an array. Again, `AShape*` indicates that it is an array of pointers to abstract shapes. Figure 1.13 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 1.15(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 1.16 shows the listing of the main program and the functions 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, prompt for the user to manipulate them, and finally to delete them.

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

```
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 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.

```
// File: Ch01/Shape/ShapeMain.cpp

#include <iostream.h> // cin, cout.
#include <stddef.h> // NULL.
#include <ctype.h> // toupper.
#include "Utilities.hpp" // iPromptBetween, dPromptGE.
#include "AShape.hpp"
#include "Line.hpp"
#include "Rectangle.hpp"
#include "Circle.hpp"
#include "RightTriangle.hpp"
#include "NullShape.hpp"
#include "ShapeMain.hpp"

void main() {
   const int NUM_SHAPES = 5;
   AShape* shapes[NUM_SHAPES];
   initialize (shapes, NUM_SHAPES);
   promptLoop (shapes, NUM_SHAPES);
   cleanUp (shapes, NUM_SHAPES);
}

void initialize (AShape* shapes[], int cap) {
   for (int i = 0; i < cap; i++) {
      shapes[i] = new NullShape;
   }
}

void cleanUp (AShape* shapes[], int cap) {
   for (int i = 0; i < cap; i++) {
      delete shapes[i];
      shapes[i] = NULL;
   };
}
```

Figure 1.16 The main program that uses the abstract and concrete classes. Functions `initialize` and `cleanUp` are shown here. The program continues on page 27.

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] = new 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 `new`. When the `new` operator 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 `new` operator allocates storage from the heap for a null shape and returns a pointer to the null shape, which is given to `shapes[i]`.

During execution of function `promptLoop`, the user may create and change many shapes. When the main program calls function `cleanUp`, `shapes` could have pointers to any combination of shapes with any possible combination of appropriate dimensions. The call to `cleanUp` returns the storage for the shapes to the heap with the statement

```
delete shapes[i];
```

The `delete` operator is the inverse of the `new` operator. When programming in C++ you should always be careful with the use of `new`. Every time you write a `new` operation you should ask yourself where the corresponding `delete` is. In this example, `new` is in function `initialize`, and the corresponding `delete` is in function `cleanUp`.

Figure 1.16 (page 27) 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[iPromptBetween (“Which shape?”,
           0, cap - 1)]);
```

The subscript of `shapes` is

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

which prompts the user for an integer between 0 and 4. If, as in Figure 1.13, the user enters the erroneous value of 7 followed by the valid value of 2, `iPromptBetween` 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 1.16 (page 28) 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

```
void promptLoop (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  ";
      cout << "(s)cale  (d)isplay  (q)uit: ";
      cin >> response;
      switch (toupper (response)) {
         case 'M':
            makeShape (shapes[iPromptBetween ("Which shape?", 0, cap - 1)]);
            break;
         case 'C':
            clearShape (shapes[iPromptBetween ("Which shape?", 0, cap - 1)]);
            break;
         case 'A':
            printArea (shapes[iPromptBetween ("Which shape?", 0, cap - 1)]);
            break;
         case 'P':
            printPerimeter (shapes[iPromptBetween ("Which shape?", 0, cap - 1)]);
            break;
         case 'S':
            scaleShape (shapes[iPromptBetween ("Which shape?", 0, cap - 1)]);
            break;
         case 'D':
            displayShape (shapes[iPromptBetween ("Which shape?", 0, cap - 1)]);
            break;
         case 'Q':
            break;
         default:
            cout << "\nPlease enter only one of the following characters: ";
            cout << "m, c, a, p, s, n, d, q." << endl;
            break;
      }
   } while (toupper (response) != 'Q');
}
```

Figure 1.16 (continued) Function `promptLoop` from the main program listing. The main program listing continues on page 28.

new rectangle. The statement

```
sh = new 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.

This assignment also illustrates the implicit call to a constructor. The existence of a constructor for the rectangle class causes the `new` operation to insert a call to the constructor after allocation from the heap. Compare the following steps to those listed on page 26 where the null shape class does not have a constructor. In this case, because the rectangle class does have a constructor, when the `new` operator executes it does three

```
void makeShape (AShape*& sh) {
   switch (shapeType ()) {
      case 'L':
         delete sh;
         sh = new Line;
         break;
      case 'R':
         delete sh;
         sh = new Rectangle;
         break;
      case 'C':
         delete sh;
         sh = new Circle;
         break;
      case 'T':
         delete sh;
         sh = new 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 1.16 (continued) Functions `makeShape` and `shapeType` from the main program listing. The main program listing continues on page 30.

thintgs. 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 `new` operator allocates storage from the heap for the attributes of `Rectangle`, which, from Figure 1.9 (page 15) are

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

Second, the `new` operator 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 1.10 (page 17) 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 `new` operator returns a pointer to newly allocated and initialized storage.

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

```
sh = new 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`. 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

```
void clearShape (AShape*& sh) {
   delete sh;
   sh = new NullShape;
}

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

void printPerimeter (AShape* sh) {
   cout << "Perimeter: " << sh->perimeter () << endl;
}

void scaleShape (AShape* sh) {
   sh->scale (dPromptGE ("Scale factor?", 0.0));
}

void displayShape (AShape* sh) {
   sh->display (cout);
}
```

Figure 1.16 (continued) Functions `clearShape`, `printArea`, `printPerimeter`, `scaleShape`, and `display` from the main program listing. The last four functions exhibit polymorphic dispatch. This concludes the main program listing

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 1.9, page 15.

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 1.16 (page 30), which shows the implementation of `printArea`, `printPerimeter`, `scaleShape`, and `display`. For example, the implementation of function `printArea` is

```
void printArea (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 π 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 {
   eLINE, eRECTANGLE, eCIRCLE,
   eRIGHT_TRIANGLE, eNULL_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 `eRECTANGLE` then `dim1` would store the length of the rectangle and `dim2` would store its width. But if `kind` had value `eCIRCLE` 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 (ShapeType* sh) {
   switch (sh->kind) {
      case eLINE:
         cout << "Area: " << lineArea (sh) << endl;
         break;
      case eRECTANGLE:
         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 (ShapeType* sh) {
   return sh->dim1 * sh->dim2
}
```

Compare this with the corresponding implementation of the function in Figure 1.10 (page 17) 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;
}
```

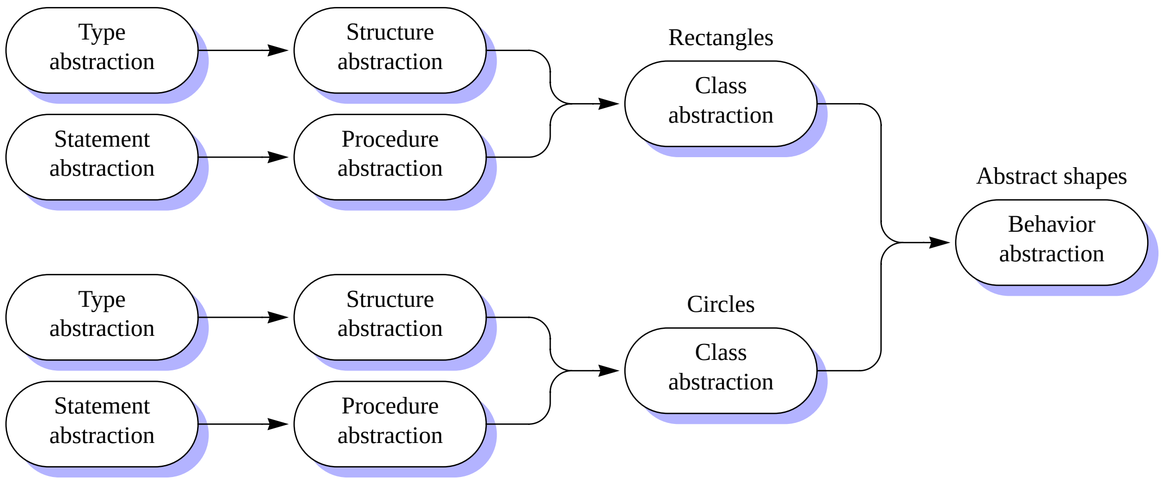


Figure 1.17 The abstraction process. Each step in the process consists of collecting many items at one layer into a single concept at the next higher layer. Type and structure abstraction are aspects of data abstraction, and statement and procedure abstraction are aspects of control abstraction. See Figure 1.1, page 2 for type abstraction, Figure 1.2, page 4 for structure abstraction, Figure 1.3, page 5 for statement abstraction, Figure 1.4, page 6 for procedure abstraction, Figure 1.5, page 7 for class abstraction, and Figure 1.7, page 11 for behavior abstraction.

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()`. How does it know which `area()` method to call? 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 1.17 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 at a high level the specification of the problem. 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.]

Exercises

This book comes with a set of software called the “distribution software”, that you should have access to 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 from the Internet at

`ftp://ftp.pepperdine.edu/pub/compsci/dp4ds/`

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 1.13. Include the precondition tests in your code.

1–4 The precondition for function `cleanUp` in `ShapeMain.h` from Figure 1.14 contains the phrase

```
Pre: shapes[0..cap - 1] is allocated,
and all elements are well-defined.
```

Rewrite the phrase “all elements are well-defined” formally using the proper quantification from the predicate calculus. Assume the predicate *wellDefined* (*s*) to indicate that *s* is well-defined.