Chapter *6*

Selection

Some problems can be solved by a fixed computation. For example, to compute the area of a rectangle you always multiply the length by the width. Many problems, however, cannot be solved by a fixed computation. For instance, some businesses sell their products at a price that depends on the quantity of the order. They charge a lower price per ball for an order of 200 golf balls than for an order of 10 golf balls. A program to calculate the total dollar amount of an order cannot simply multiply the quantity by a fixed unit price if the unit price itself depends on the quantity.

This chapter describes boolean expressions and IF statements. Together these features of the Component Pascal language permit the programmer to alter, or select, the computation depending on the outcome of a test on one or more data values.

Boolean expressions and types

Boolean expressions always have one of two values, either true or false. The simplest boolean expressions use the relational operators of Figure 6.1. In mathematics notation, the "less than or equal to" operator is ≤. This symbol is not available on most keyboards, so Component Pascal programs require that you write "less than or equal to" as the two symbols <= without a space between them. The same idea applies to the "greater than or equal to" operator. The "not equal to" sign # resembles the mathematical symbol ≠.

Operator	Meaning
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
#	Not equal to

Figure 6.1
The relational operators.

Example 6.1 An example of a boolean expression is

```
income > 2400
```

where income is an integer variable. This expression is either true or false, depending on the value of income. If income has the value 2500, the expression is true. If it has the value 2300 or even 2400, the expression is false. ▮

Example 6.2 In contrast to the previous example, the expression

```
income >= 2400
```

evaluates to true if income has the value 2400. ▮

Variables of type boolean are declared in the variable declaration part similarly to the way numeric and character variables are declared. A boolean variable can have one of two values, true or false.

Example 6.3 The following variable declaration part declares rich to be a boolean variable.

```
VAR
    rich: BOOLEAN;
```

The assignment statement

```
rich := income > 2400
```

gives rich the value true if income has value greater than 2400, and gives the value false otherwise. ▮

The ODD function is a built-in Component Pascal function that takes an integer parameter and returns true if the value of the integer is odd. There is no corresponding even function.

Example 6.4 Suppose that i is a variable of type integer that has the value 14. Then the boolean expression

```
ODD(i)
```

has the value false, and the expression

```
ODD(i + 1)
```

has the value true. ▮

Boolean expressions may contain the AND operator, written &. Suppose *p* is the statement “The sky is green,” which is obviously false, and *q* is the statement “Computer science is fun," which is obviously true. Then, *p* & *q* is the statement "The sky is green and computer science is fun," which is false. For the entire statement *p* & *q* to be true, *p* must be true apart from *q*, and *q* must be true apart from *p*. If either or both are false, then the entire statement is false. Figure 6.2(a), the truth table for the & operator, summarizes these ideas.

Boolean expressions may also contain the OR operator, written OR. With *p* and *q* representing the same statements about the sky and computer science, *p* OR *q* is the statement "The sky is green or computer science is fun." This time, the entire statement is true. *p* OR *q* is true if *p* is true, if *q* is true, or if they are both true. Figure 6.2(b), the truth table for the OR operator, summarizes these ideas.

One other boolean operator is the NOT operator, written ~. If *p* is the statement "The sky is green," which is false, then ~ *p* is the statement "The sky is not green," which is true. Figure 6.2(c) is the truth table for the ~ operator.

Figure 6.2 Truth tables for the boolean operators.

p	***q***	***p* & *q***
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

(a) The & operator.

p	***q***	***p* OR *q***
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

(b) The OR operator.

p	**~ *p***
TRUE	FALSE
FALSE	TRUE

(c) The ~ operator.

Example 6.5 Suppose that i is a variable of type integer that has the value 8. Then the boolean expression

```
~ODD(i)
```

has the value true. ■

Sometimes it is possible to simplify a boolean expression that contains the ~ operator with a relational operator.

Example 6.6 The boolean expression

```
~(numSides > 8)
```

first evaluates the boolean expression (numSides > 8). If numSides has the value 10, then (numSides > 8) is true, and ~(numSides > 8) is false. A simpler way to write an equivalent boolean expression is

```
numSides <= 8
```

Suppose again that numSides has the value 10. Then numSides <= 8 is false, as it was in the previous expression. The two boolean expressions are the same regardless of the value of numSides. ■

This example demonstrates that the <= operator is the inverse of the > operator. Figure 6.3 shows the relational operators and their inverses.

Operator	Inverse Operator
=	#
<	>=
<=	>

Figure 6.3 The inverses of the relational operators.

Example 6.7 You could write the expression

```
~(numTrials = maxTrials)
```

more simply as

```
numTrials # maxTrials
```

because the # operator is the inverse of the = operator. ▮

A common mistake you should avoid is putting the ~ operator next to a relational operator. A relational operator must be placed between two integers, reals, or characters and cannot be next to a ~.

Example 6.8 The compiler will not accept the expression

```
numTrials ~= maxTrials
```

because the equals operator cannot have the NOT operator to its left. ▮

Another error you should avoid is combining two relational operators with one variable, as is frequently done in mathematics. The mathematical expression

$5 \leq n < 10$

means that n is greater than or equal to 5 and less than 10. Such expressions are common in mathematics. However, they are illegal in Component Pascal.

Example 6.9 To test if the variable numTrials is greater than or equal to 5 and less than 10, you may be tempted to write the boolean expression as

```
5 <= numTrials < 10
```

This expression is illegal, because the same operand, numTrials, is used by both operators. You should write it with the & operator as follows:

```
(5 <= numTrials) & (numTrials < 10)
```

If numTrials has the value 6, this boolean expression evaluates to true AND true, which is true.

Two other rules, known as De Morgan's laws, can sometimes help to simplify boolean expressions. Suppose that *p* and *q* are boolean expressions. De Morgan's laws are

$\sim(p \text{ OR } q) = \sim p \,\&\, \sim q$
$\sim(p \,\&\, q) = \sim p \text{ OR } \sim q$

Example 6.10 As an example of the first of De Morgan's laws, you can write the boolean expression

```
~((slope >= 1.0) OR (length <= 0.0))
```

more simply as

```
(slope < 1.0) & (length > 0.0)
```

because < is the inverse of >=, and > is the inverse of <=.

When the Component Pascal compiler encounters a boolean expression in your program, it gives the NOT operator the highest precedence, the AND operator the next highest precedence, and the OR operator the lowest precedence of the three. Figure 6.4 summarizes these precedence rules and also compares the precedence of the boolean operators to the relational and arithmetic operators.

Operator	Precedence
~	Highest
&, DIV, MOD, /, *	
OR, +, -	
=, #, <, >, <=, >=	Lowest

Figure 6.4 Precedence of the Component Pascal operators.

Example 6.11 The Component Pascal compiler interprets the boolean expression

```
~ p & q
```

as (~p) & q rather than ~(p & q).

Example 6.12 If alpha, beta, and gamma are integer variables, the boolean expression

```
alpha < beta & gamma = 0
```

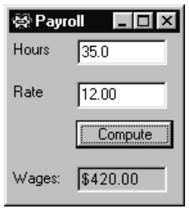
is illegal because the compiler groups beta & gamma first. The & operator expects boolean operands, but beta and gamma are integers. You should write the expression as

```
(alpha < beta) & (gamma = 0)
```

which is now a legal boolean expression. ■

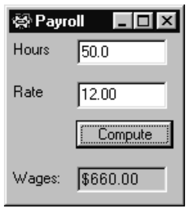
IF statements

IF statements allow you to solve problems that are not based on fixed computations. The idea is to evaluate a boolean expression, and if that expression is true, perform a computation. Figure 6.5 shows an example of such a conditional computation. The dialog box is for computing the wages for an employee who may have worked overtime. Customarily, weekly wages are computed as the hourly rate times the number of hours worked, as long as the employee does not work more than 40 hours. If the employee works more than 40 hours, then the number of hours in excess of 40 are paid at time and a half. That is, the hourly rate for those hours beyond 40 is 1.5 times the normal rate.


Payroll
Hours: 35.0
Rate: 12.00

Wages: $420.00

(a) Without overtime.


Payroll
Hours: 50.0
Rate: 12.00

Wages: $660.00

(b) With overtime.

Figure 6.5
The dialog box for a payroll calculation.

The listing in Figure 6.6 shows an implementation of the dialog box for calculating the payroll. The assignment statement

```
wages := d.hours * d.rate
```

computes wages as the product of d.hours and d.rate assuming no overtime. When the input is 50 for the hours worked and 12 for the hourly rate as shown in Figure 6.5(b), wages gets the value 600.00. This value is not yet correct because the 10 hours beyond 40 were computed at straight time, not time and a half.

```
MODULE Pbox06A;
    IMPORT Dialog, PboxStrings;
    VAR
        d*: RECORD
            hours*, rate*: REAL;
            message-: ARRAY 32 OF CHAR
        END;

    PROCEDURE ComputeWages*;
        VAR
            wages: REAL;
            wageString: ARRAY 32 OF CHAR;
    BEGIN
        wages := d.hours * d.rate;
        IF d.hours > 40.0 THEN
            wages := wages + (d.hours - 40.0) * 0.5 * d.rate
        END;
        PboxStrings.RealToString(wages, 1, 2, wageString);
        d.message := "$" + wageString;
        Dialog.Update(d)
    END ComputeWages;

BEGIN
    d.hours := 0.0; d.rate := 0.0;
    d.message := ""
END Pbox06A.
```

Figure 6.6
A payroll calculation program. It uses an IF statement without an ELSE part.

The words IF and THEN are Component Pascal reserved words. When an IF statement executes, it first evaluates the boolean expression following the reserved word IF. If the boolean expression is true, it executes the statement sequence between the reserved word THEN and the reserved word END. Otherwise, it skips the statement sequence. In this example, after Wages is computed as d.hours * d.rate, the IF statement evaluates the boolean expression

```
d.hours > 40.0
```

which is true, because the value of d.hours is 50. So the assignment statement following the IF statement

```
wages := wages + (d.hours - 40.0) * 0.5 + d.rate
```

executes. The value of wages is increased to reflect the extra amount (at half time) earned in overtime.

Appendix A shows that the EBNF syntax for an IF statement is

```
IF Expr THEN StatementSeq {ELSIF Expr THEN StatementSeq} [ELSE StatementSeq] END
```

The IF statement in Listing 6.6 does not have an ELSIF part, nor does it have an ELSE part.

Flowcharts

You can visualize the action of an IF statement with a flowchart. Figure 6.7 shows some of the more common flowchart symbols. The start symbol corresponds to the reserved word BEGIN, which starts the executable statements of a Component Pascal procedure. The stop symbol, which is the same shape as the start symbol, corresponds to the reserved word END. The parallelogram corresponds to input and output statements. Rectangles correspond to processing, performed by the assignment statement in Component Pascal. The hexagon is a symbol that indicates the test of some condition. It is used in several Component Pascal statements, including the IF statement. The symbol that looks like a hamburger corresponds to the CASE statement, a selection statement described later in this chapter. The circle is the collector symbol for joining lines from other flowchart symbols.

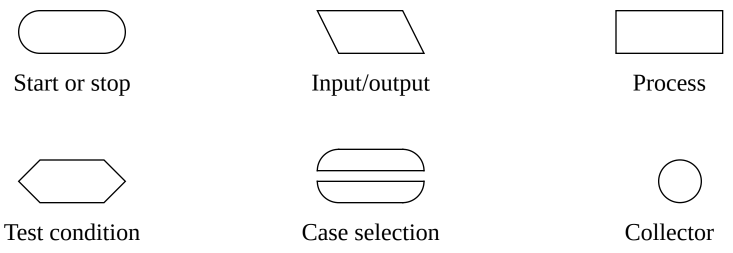


Figure 6.7
The flowchart symbols.

Figure 6.8 is the flowchart for an IF statement without an ELSE part. The incoming arrow from the top points to the test condition box, which represents the boolean expression of the IF statement. If the boolean expression is true, control branches to the left to the processing box, which represents the statement sequence after the reserved word THEN. If the boolean expression is false, control branches to the right, skipping execution of the statement sequence after the reserved word THEN. The two branches of the IF statement join at the collector symbol corresponding to the reserved word END. Figure 6.9 is the program of Figure 6.6 in flowchart form.

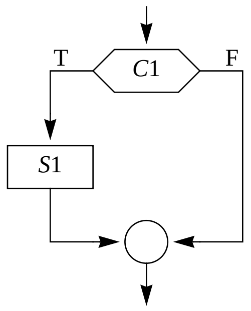


Figure 6.8
The flowchart for an IF statement without an ELSE

Flowcharts are useful for visualizing the logic of a program. They used to be considered helpful in software design, but have fallen out of favor for several reasons. Flowcharts are fine for small programs but they require huge pages of paper for large programs. They also require artwork and are consequently more difficult to modify than the programs they represent. This book presents flowcharts to help you visualize the behavior of some Component Pascal statements. As you gain experience writing Component Pascal programs, however, you will not need to rely on flowcharts to design your software.

IF statements with an ELSE part

The listing in Figure 6.10 presents a different way to compute the wage correctly. Its output is identical to the output of Figure 6.6.

The program uses an IF statement with an ELSE part. If the boolean expression in the IF statement is true, the statement sequence following the THEN part

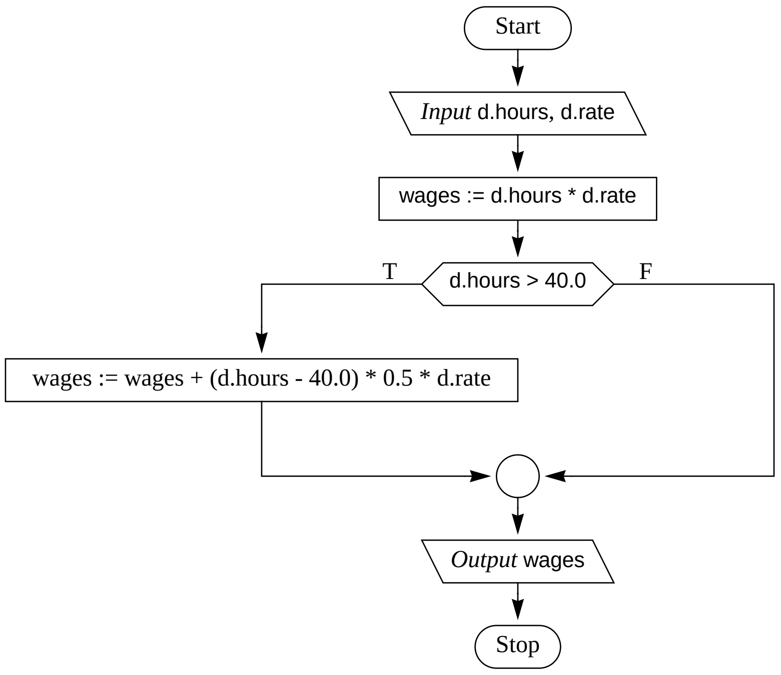

Start
Input d.hours, d.rate
wages := d.hours * d.rate
T
d.hours > 40.0
F
wages := wages + (d.hours - 40.0) * 0.5 * d.rate
Output wages
Stop

Figure 6.9
The flowchart for the program of Figure 6.6.

```
wages := d.hours * d.rate
```

executes. After it executes, the statement sequence after the reserved word ELSE

```
wages := 40.0 * d.rate + (d.hours - 40.0) * 1.5 * d.rate
```

is skipped. If, on the other hand, the boolean expression in the IF statement is false, the THEN part is skipped, and the ELSE part executes. The effect of the IF statement is to select one of the two statements to execute.

There is no semicolon after the statement following the reserved word THEN. You can see in Appendix A that there are no semicolons in the syntax definition for an IF statement. Then why is there a semicolon after the reserved word END? That semicolon is there to separate the entire IF statement from the following PboxStrings.RealToString. These statements are part of the statement sequence of the body of the procedure. The EBNF description of a procedure declaration is

PROCEDURE [Receiver] IdentDef [FormalPars] “ ; ” DeclSeq [BEGIN StatementSeq] END Ident.

and the EBNF description of a statement sequence is

Statement {“ ; ” Statement}.

```
MODULE Pbox06B;
    IMPORT Dialog, PboxStrings;
    VAR
        d*: RECORD
            hours*, rate*: REAL;
            message-: ARRAY 32 OF CHAR
        END;

    PROCEDURE ComputeWages*;
        VAR
            wages: REAL;
            wageString: ARRAY 32 OF CHAR;
    BEGIN
        IF d.hours <= 40.0 THEN
            wages := d.hours * d.rate
        ELSE
            wages := 40.0 * d.rate + (d.hours - 40.0) * 1.5 * d.rate
        END;
        PboxStrings.RealToString(wages, 1, 2, wageString);
        d.message := "$" + wageString;
        Dialog.Update(d)
    END ComputeWages;

BEGIN
    d.hours := 0.0; d.rate := 0.0;
    d.message := ""
END Pbox06B.
```

Figure 6.10
A payroll calculation program that uses an IF statement with an ELSE part.

So, the statements in the statement sequence between the BEGIN and END of the procedure are separated by semicolons. We now have three general rules for placing semicolons:

- Do not place a semicolon after a THEN.
- Do not place a semicolon before an END.
- Do not place a semicolon before an ELSE.

Figure 6.11 shows the flowchart for an IF statement with an ELSE part. It is similar to the flowchart for an IF statement without an ELSE part in two respects. Both flowcharts have exactly one collector, and both have exactly one arrow coming in at the top and one arrow going out at the bottom.

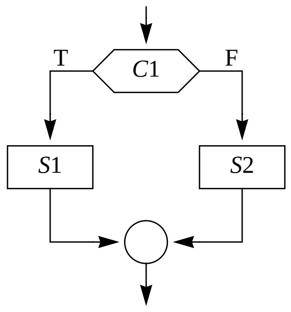
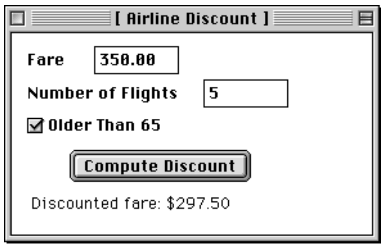


Figure 6.11
The flowchart for an IF statement with an ELSE part.

Boolean variables

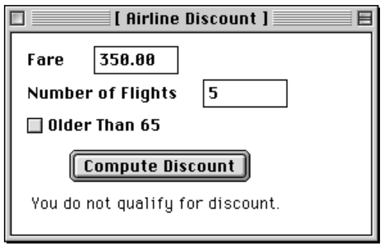
Figure 6.12 shows the dialog box for the next program. The program determines whether a customer qualifies for a 15% airline discount. If the customer qualifies, it computes the discounted fare. Otherwise, it states that the customer does not qualify. A customer qualifies by having made more than 4 flights during the previous 12 months and being 65 years of age or older.

The dialog contains a control called a check box. The user can check the square



Discounted fare: $297.50

(a) Fare with discount.



You do not qualify for discount.

(b) Fare without discount.

Figure 6.12 The dialog box for computing a possible discount for an airline fare.

input field to indicate whether she is older than 65. It is clear that the input field for Fare is linked to a variable of type real, and the input field for the number of flights is linked to a variable of type integer. But what is type of the variable to which the check box is linked? There are two possibilities for the input of this field. Either the box is checked or it is not checked. So, the variable to which it is linked has type boolean. If the box is checked the boolean variable gets the value true, and if it is not checked the variable gets the value false. The listing in Figure 6.13 shows the program for this dialog box.

Constants

The constant section is similar to the variable section, except that an equal sign follows the identifier instead of a colon. Another difference is that in the variable section a type is associated with each identifier, while in the constant section a value is associated with each identifier. Constants are similar to variables in that you refer to them by their names, which are Component Pascal identifiers. However, you cannot change the value of a constant the way you can the value of a variable.

Example 6.13 The assignment statement

```
discount := 0.20
```

would be illegal in this program, because discount is a constant. ▮

Procedure FlightDiscount defines the identifier discount to be the constant 0.15 and flightLimit to be 4. The program would produce exactly the same result without the constant section and with the expression in the IF statement changed from

```
IF (d.numFlights > flightLimit) & d.olderThan65 THEN
```

to

```
IF (d.numFlights > 4) & d.olderThan65 THEN
```

and the computation for the fare changed from

```
fare := (1.0 - discount) * d.fare;
```

```
MODULE Pbox06C;
    IMPORT Dialog, PboxStrings;
    VAR
        d*: RECORD
            fare*: REAL;
            numFlights*: INTEGER;
            olderThan65*: BOOLEAN;
            message-: ARRAY 64 OF CHAR
        END;

    PROCEDURE FlightDiscount*;
        CONST
            discount = 0.15;
            flightLimit = 4;
        VAR
            fare: REAL;
            fareString: ARRAY 32 OF CHAR;
    BEGIN
        IF (d.numFlights > flightLimit) & d.olderThan65 THEN
            fare := (1.0 - discount) * d.fare;
            PboxStrings.RealToString(fare, 1, 2, fareString);
            d.message := "Discounted fare: $" + fareString
        ELSE
            d.message := "You do not qualify for discount."
        END;
        Dialog.Update(d)
    END FlightDiscount;

BEGIN
    d.fare := 0.0; d.numFlights := 0;
    d.olderThan65 := FALSE;
    d.message := ""
END Pbox06C.
```

Figure 6.13
A program to compute the discount on an airline ticket.

to

```
fare := 0.85 * d.fare;
```

The advantages of a constant

So, what is the advantage of a constant section? One advantage is the ease with which you can modify the program. This program is short, and it is easy to locate the assignment statement where it computes the discount. If you wanted to modify the program to change the discount to 20% instead of 15%, you could find the assignment statement with your text editor and change the 0.85 to 0.80. But in a large program, the statement that performs the computation may be difficult to locate. Also, more than one computation may need to be modified to make one change.

For example, suppose you write a big tax computation program in which a tax rate for both businesses and individuals is 20%. These rates are used in many different computations. You do not use a constant definition part, so that the value 0.20 is scattered in various expressions throughout the program. Now suppose that a new tax law changes the rate for businesses to 30% but leaves the rate for individuals unchanged. To modify the program you cannot simply use your text editor to change every occurrence of 0.20 to 0.30, because that would change the rate for individuals as well.

On the other hand, suppose you write the program with a constant section that defines

```
CONST
   businessRate = 0.20;
   individualRate = 0.20;
```

and use these identifiers in the appropriate expressions in the program. Then, if the tax law changes the business rate to 30% you only need to change one value at the beginning of the program to modify it correctly.

Another advantage of constants is the increased readability that identifiers provide. In this program, the expression

```
(1.0 - discount) * d.fare
```

represents the meaning of the computation better than the expression

```
0.85 * d.fare
```

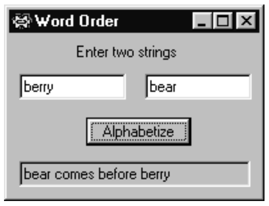
The presence of identifier discount tells the reader explicitly that a discounted fare is being computed.

Selection with strings

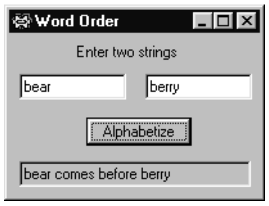
The alphabetizing algorithm

Component Pascal provides a convenient feature for testing strings according to alphabetic order. Consider the problem to determine whether string “berry” comes before “bear” in alphabetic order. The proper algorithm compares the first two letters. Because the first b in “berry” equals the first b in “bear”, you must compare the second letters. Alas, the e in “berry” equals the e in “bear” as well, so you must go to the third letter. Finally, the third letters are not equal. Because a is less than r, as shown on the number line for the character values in Figure 4.11, “bear” is less than “berry” and so comes first in alphabetic order regardless of the letters beyond the third. Of course, the algorithm must handle words of unequal length as well. Which comes first in alphabetic order, “batter” or “bat”?

With Component Pascal, you can use the relational operators of Figure 6.1 to compare not only integers, reals, and individual characters, but strings of characters as well. The comparison of two strings takes complete account of the above algorithm, including the case where two strings are of unequal length. Figure 6.14 shows a dialog box, which is implemented by the program of Figure 6.15, to compare two strings entered by the user. Regardless of whether the user enters “bear” or “berry” first, the output indicates which comes first in alphabetic order. It also correctly handles the case of unequal word lengths.



(a) First “berry”, then “bear”.



(b) First “bear”, then “berry”.

Figure 6.14 A dialog box for comparing strings in alphabetic order.

```
MODULE Pbox06D;
    IMPORT Dialog;
    VAR
        d*: RECORD
            string1*, string2*: ARRAY 16 OF CHAR;
            message-: ARRAY 64 OF CHAR;
        END;

    PROCEDURE Alphabetize*;
    BEGIN
        IF d.string1 < d.string2 THEN
            d.message := d.string1 + " comes before " + d.string2
        ELSE
            d.message := d.string2 + " comes before " + d.string1
        END;
        Dialog.Update(d)
    END Alphabetize;

BEGIN
    d.string1 := ""; d.string2 := ""
END Pbox06D.
```

Figure 6.15 A program to compare two strings.

Although this program correctly alphabetizes strings whose letters are all lowercase or all uppercase, it will not always work correctly if the user enters strings that contain both upper- and lowercase letters. Figure 6.16 shows what happens when the user enters “bear” and “Berry”. The output erroneously claims that “Berry” comes before “bear”. The origin of the problem lies in the ordering of the characters on the number line in Figure 4.11. All the uppercase characters lie to the left of all the lowercase characters. Therefore, the character B is less than the character b, and the alphabetizing algorithm blindly concludes that “Berry” is less than “bear” without even considering the characters beyond the first one in the string. Because every uppercase letter is less than every lowercase letter, the program will even claim that “Zebra” comes before “antelope”!

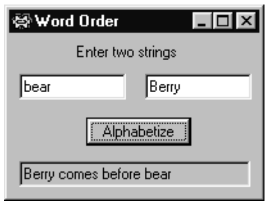


Figure 6.16 Erroneous output from the program of Figure 6.15.

This program needs to be improved, and it can be with the help of module PboxStrings, whose interface is listed in Figure 4.12. The module contains procedure ToLower, which is listed as

```
PROCEDURE ToLower (from: ARRAY OF CHAR; OUT to: ARRAY OF CHAR);
```

You supply an actual parameter for from that has already been given a string value, and a variable for to that is initially undefined. When you call ToLower, the procedure will copy all the characters from from to to, converting any uppercase letters to lowercase. Recall that OUT specifies call by result, which has the effect of changing your actual parameter.

How does this procedure help solve the problem? You can declare two local variables, say lower1 and lower2, in procedure Alphabetize and use ToLower to give them the lowercase versions of d.string1 and d.string2 respectively. In the IF statement, compare lower1 with lower2 instead of d.string1 with d.string2. Because lower1 and lower2 will contain only lowercase characters, the comparison will be correct. Of course, all this manipulation should take place behind the scenes unknown to the user. If the user enters "bear" and "Berry" the message on the dialog box should read

```
Berry comes before bear
```

and not

```
berry comes before bear
```

Implementation of this improvement is a problem at the end of the chapter.

Using IF statements

This section points out some aspects of IF statements that tend to give beginning programmers problems. Some are style guidelines that have been mentioned previously, while others are unique to IF statements. In the following discussion and throughout the remainder of the book, we will sometimes use the word *code*. One meaning for code is what a programmer writes in a program listing. Coding an algorithm means writing a program in some programming language that will execute the algorithm on a computer. A code fragment is a few lines of code from a program listing.

The word code

A common tendency with boolean variables is to use a redundant computation with the equals operator. A boolean variable is a special case of a boolean expression, and so can be used alone as a boolean expression in an IF statement.

Example 6.14 In the listing of Figure 6.13, you could write the test for the IF statement as

```
IF (d.numFlights > flightLimit) & (d.olderThan65 = TRUE) THEN
```

With this test the program still works correctly, because the expression d.olderThan65 = TRUE evaluates to true when d.olderThan65 has the value true and to false when d.olderThan65 has the value false. But this is bad style because it contains a redundant computation. The more straightforward test

```
IF (d.numFlights > flightLimit) & d.olderThan65 THEN
```

presented in Listing 6.13 is better. ▮

Boolean variables are useful because they allow Component Pascal IF statements to be written similar to English phrases whose meaning is close to the effect of the Component Pascal statement. In the previous example, IF (d.numFlights > flightLimit) & d.olderThan65 THEN is much like an English phrase. You should name your boolean variables so that the test of an IF statement corresponds to the way you would phrase the test in English.

Example 6.15 Suppose that exempt is a boolean variable that indicates whether a taxpayer is exempt from a tax. Instead of writing the test

```
IF exempt = FALSE THEN
```

you should write the equivalent test

```
IF ~exempt THEN
```

because this corresponds more closely to the way you would state the test in English. ▮

It is worth repeating a point here that was made in a previous chapter: do not save typing time by choosing extremely short identifiers at the expense of program readability.

Example 6.16 In the previous example, if you choose e for the identifier instead of exempt, the test of the IF statement becomes

```
IF ~e THEN
```

which would be more difficult for a human reader to understand. ▮

Our last problem area concerns the unnecessary duplication of code. Suppose you write an IF statement with an ELSE part that has the following form:

```
IF Condition 1 THEN
    Statement 1 ;
    Statement 2
ELSE
    Statement 3 ;
    Statement 2
END
```

where Statement 2 is the same statement in both alternatives of the IF statement. Condition 1 is a boolean expression. If it is true, Statement 1 executes, followed by Statement 2. Otherwise, Statement 3 executes, followed by Statement 2. Regardless of whether Condition 1 is true or false, Statement 2 executes. It is simpler to write

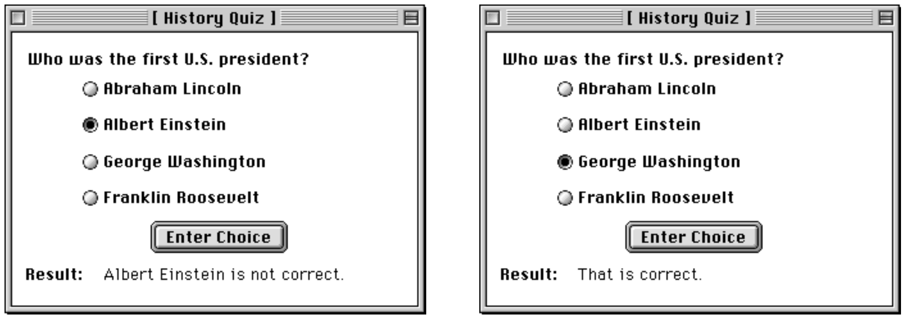
```
IF Condition 1 THEN
    Statement 1
ELSE
    Statement 3
END;
Statement 2
```

which executes like the previous code but does not duplicate Statement 2 in the code fragment.

Radio buttons

Each test in an IF statement evaluates a boolean expression, which can have the values true or false. So every test selects between two alternatives. A single CASE statement, however, can select between more than just two alternatives. The type of the expression to be tested cannot be a boolean because more than two alternatives are possible. Its type is usually integer.

Radio buttons are common controls in dialog boxes when the user is required to make one of several choices. They are a generalization of check boxes, which require that the user select one of two choices—either checked or not checked. Radio buttons always come in sets of two or more. Figure 6.17 shows a dialog box with four radio buttons. The dialog box requests that the user answer a multiple choice question about U.S. history. It provides four choices, only one of which is correct. A CASE statement is an appropriate way to analyze the input from this set of radio buttons because more than two alternatives is possible.



[History Quiz]

Who was the first U.S. president?

○ Abraham Lincoln
● Albert Einstein
○ George Washington
○ Franklin Roosevelt

Enter Choice

Result: Albert Einstein is not correct.

[History Quiz]

Who was the first U.S. president?

○ Abraham Lincoln
○ Albert Einstein
● George Washington
○ Franklin Roosevelt

Enter Choice

Result: That is correct.

Figure 6.17
A dialog box with a set of four radio buttons.

The controls are called radio buttons because of their similarity to the push buttons on old automobile radios for tuning in stations. You can only have one station tuned in at a time. If you change the setting to a new radio station by pushing a button, then any button that was previously pressed is released. Similarly, in a dialog box with radio buttons if one button is pressed, as indicated by the solid circle, then any button that was previously pressed is released, as indicated by the open circles.

Until now, each input/output control in a dialog box has been linked to a variable in an exported record that has consistently been named d. It would appear from Figure 6.17 that we would need five fields in d—one for each of the four radio buttons and one for the result. Such is not the case, however. Instead, all four radio buttons are linked to a single integer variable in d. The listing in Figure 6.18 shows this integer as multipleChoice.

```
MODULE Pbox06E;
    IMPORT Dialog;
    VAR
        d*: RECORD
            multipleChoice*: INTEGER;
            message-: ARRAY 64 OF CHAR
        END;

    PROCEDURE PresidentQuiz*;
    BEGIN
        CASE d.multipleChoice OF
        0:
            d.message := "Abraham Lincoln is not correct." |
        1:
            d.message := "Albert Einstein is not correct." |
        2:
            d.message := "That is correct." |
        3:
            d.message := "Franklin Roosevelt is not correct."
        END;
        Dialog.Update(d)
    END PresidentQuiz;

BEGIN
    d.multipleChoice := 0;
    d.message := ""
END Pbox06E.
```

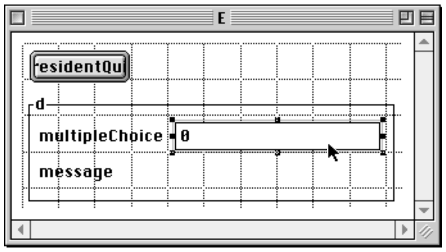
Figure 6.18
A module that takes its input from a set of radio buttons. It uses a CASE statement.

The radio buttons are linked such that when the button for Abraham Lincoln is pressed the value of d.multipleChoice is 0, when the button for Albert Einstein is pressed its value is 1, when the button for Washington is pressed its value is 2, and when the button for Roosevelt is pressed its value is 3.

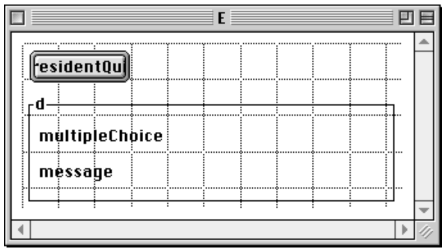
It is a bit cumbersome to set up the links for radio buttons compared to setting up the links for the other controls. The problem is that when you declare multipleChoice to be an integer in record d and create a new form linked to it by choosing Controls→New Form… the BlackBox system will supply an integer input field instead of a set of four radio buttons. You must delete the integer input field, insert four radio buttons, then set up their proper links manually. Figure 6.19 shows this process.

Figure 6.19(a) shows the controls that are provided by the BlackBox forms generator when you create a new form by selecting Controls→New Form… and link it to Pbox09A.d. The forms generator inserts an integer input field for the integer d.multipleChoice. Part (b) of the figure shows the result of selecting the input field and deleting it by pressing the delete key. Part (c) shows how to enlarge the forms document to make room for the radio buttons. Choose Edit→Select Document to select the forms document for enlarging. Part (d) shows the result of choosing Controls→Insert Radio Button. A radio button control is inserted in the forms document

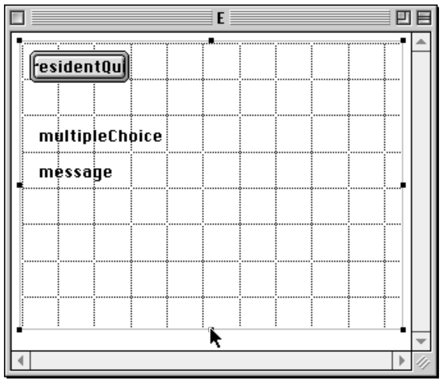
Figure 6.19
The process of constructing a dialog box with four radio buttons.



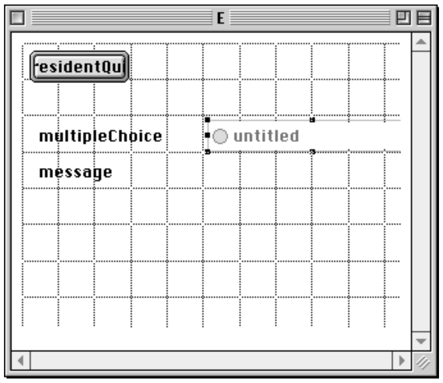
(a) Select the integer field provided by the forms generator.



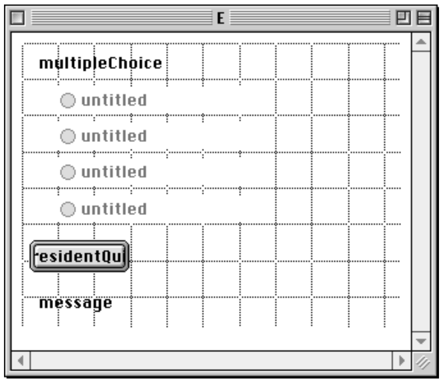
(b) Delete the integer field by pressing the delete key.



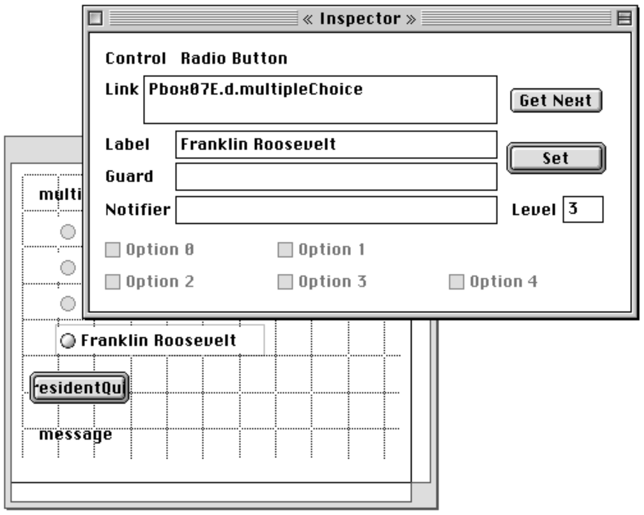
(c) Select the document and enlarge it to make room for the radio buttons.



(d) Insert the first radio button by selecting Controls→Insert Radio Button.



(e) Insert three more radio buttons and arrange them.



Control Radio Button
Link Pbox07E.d.multipleChoice
Label Franklin Roosevelt
Guard
Notifier
Level 3

(f) Set the proper links manually with the component Inspector.

at some random location. Part (e) shows the dialog box after arranging the radio button where you want it and inserting and arranging the other three radio buttons.

The last step is to set the proper links with the component inspector as shown in Figure 6.19(f). The figure shows the settings for the fourth radio button. You must manually enter the link to the exported integer, which in this case is Pbox09A.d.mutipleChoice. Enter the text that you want to appear next to the radio button in the label field, which in this case is Franklin Roosevelt. Because you want the value of d.multipleChoice to be 3 when this radio button is pushed, you must enter 3 for the level of the control as shown in the inspector in part (f).

The CASE statement

The listing in Figure 6.18 shows how the CASE statement can select from more than just two alternatives. The Component Pascal syntax for a CASE statement is

```
CASE Expr OF Case {“ | ” Case} [ELSE StatementSeq] END
```

In this program, the expression Expr between reserved words CASE and OF is simply d.multipleChoice. Following OF is a list of one or more cases, each case separated by a vertical bar |. This program has four cases separated by three vertical bars. It does not have the optional ELSE part.

The Component Pascal syntax for an individual Case is

```
[CaseLabels {“ , ” CaseLabels} “ : ” StatementSeq]
```

Each Case consists of one or more CaseLabels separated by commas followed by a colon followed by our familiar StatementSeq. In this program, the case label for the second case is 1, and the statement sequence is the single assignment statement

```
d.message := “Albert Einstein is not correct.”
```

When the user selects one of the radio buttons in the dialog box of Figure 6.17 and presses the button labeled Enter Choice, procedure PresidentQuiz executes. The first statement in the procedure is the CASE statement, which evaluates the expression d.multipleChoice. Execution then skips directly to the statement sequence of the corresponding case label skipping all other cases. Following execution of that statement sequence, execution skips directly to the end of the CASE statement again skipping all other cases. In this program, suppose the user has clicked the fourth radio button corresponding to Roosevelt. This selection causes d.mutipleChoice to have the value 3. When the CASE statement executes, it skips directly to the assignment statement

```
d.message := “Franklin Roosevelt is not correct.”
```

skipping all other cases.

If the value of the expression does not occur as a label of any case, the statement sequence following the ELSE is selected, if there is an ELSE. Otherwise, the program is aborted with a trap. This program does not require an ELSE part if the levels for the radio buttons are set up correctly. Because each level is set to have a value of 0, 1, 2, or 3 we are guaranteed that d.multipleChoice can have no other value.

Figure 6.20 shows the flowchart for the module in Figure 6.18. Note how the case selection symbol has more than one alternative arrow leaving it compared to the IF hexagon flowchart symbol that always has two arrows leaving it.

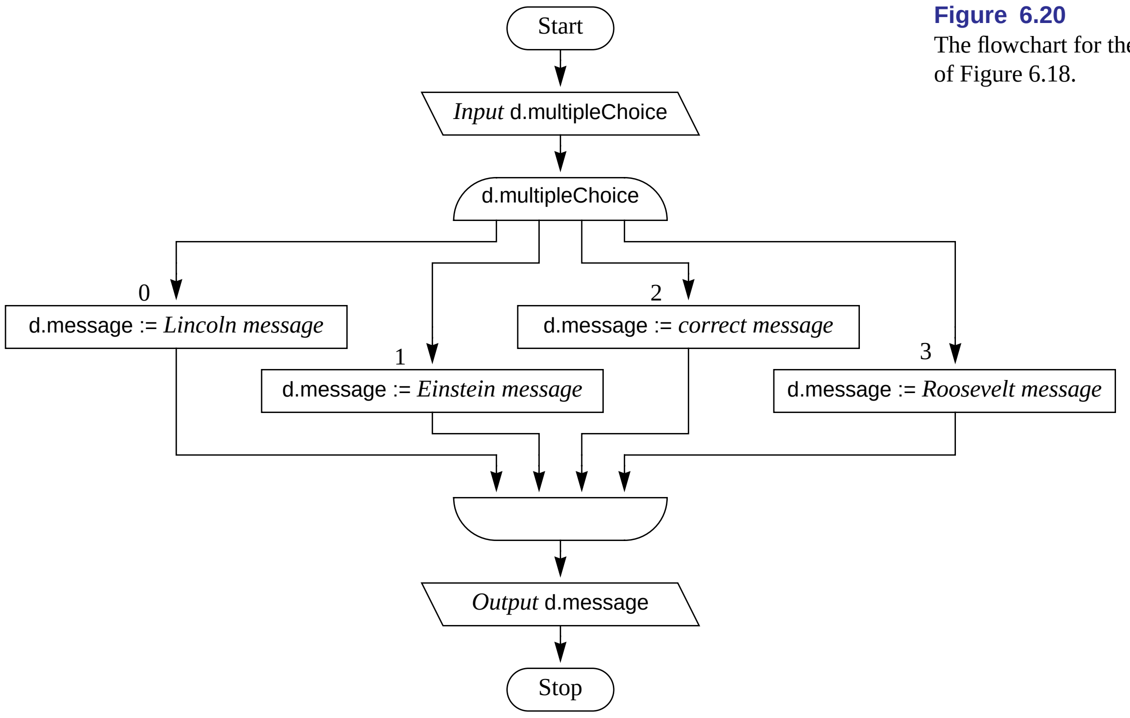

Start
Input d.multipleChoice
d.multipleChoice
0
d.message := *Lincoln message*
1
d.message := *Einstein message*
2
d.message := *correct message*
3
d.message := *Roosevelt message*
Output d.message
Stop

Figure 6.20
The flowchart for the module of Figure 6.18.

★ Guarded commands

The Component Pascal (CP) operations of &, OR, and ~ correspond directly to the Guarded Command Language (GCL) operators of conjunction, disjunction, and negation as Figure 6.21 shows. Not only are the symbols different but the precedence is different as well. In CP, ~ has the highest precedence, & has the second highest, and OR has the lowest of the three. But in GCL, $\wedge$ and $\vee$ have the same precedence, although $\neg$ has higher precedence than both as in CP. Consequently, you must be careful to place parentheses in a GCL statement where you might not need it in a CP one.

CP	GCL
&	$\wedge$
OR	$\vee$
~	$\neg$

Figure 6.21
The boolean operators in GCL.

Example 6.17 The CP boolean expression

```
(sum < 100) OR (a < b) & (b < c)
```

causes the & operation to execute first followed by the OR operation. However, the equivalent boolean expression in GCL is

$(sum < 100) \vee ((a < b) \wedge (b < c))$

where the parentheses are required to indicate that the $\wedge$ operation occurs first. ▮

The guarded command language is so called because several of its statements, including the **if** statement, contain a phrase of the form $B \rightarrow S$ known as a guarded command. B is a boolean condition that must be true in order for statement S to execute. In the same way that semicolons separate statements, the symbol [] separates guarded commands. In CP, every IF statement terminates with an END. In GCL, every **if** statement terminates with **fi**, which is **if** spelled backward. Another difference between GCL and CP is that there is no phrase corresponding to ELSE in GCL. Instead, a guarded command is used for the ELSE part. If there is no ELSE part in the CP statement, you must use a guarded command with the **skip** statement, which does nothing when it executes.

Example 6.18 In Figure 6.6, using *w* for wages, *h* for d.hours, and *r* for d.rate the CP statements

```
IF d.hours > 40.0 THEN
   wages := wages + (d.hours - 40.0) * 0.5 * d.rate
END
```

are written in GCL as

if $h > 40.0 \rightarrow w := w + (h - 40.0) * 0.5 * r$
[] $h \leq 40.0 \rightarrow$ **skip**
fi ▮

If there is an ELSE part in the CP statement, you still specify what happens in the false alternative with a guarded command in GCL.

Example 6.19 In Figure 6.7, again using *w* for wages, *h* for d.hours, and *r* for d.rate the CP statements

```
IF d.hours <= 40.0 THEN
   wages := d.hours * d.rate
ELSE
   wages := 40.0 * d.rate + (d.hours - 40.0) * 1.5 * d.rate
END
```

are written in GCL as

if $h \leq 40.0 \rightarrow w := d * r$
[] $h > 40.0 \rightarrow w := 40.0 * r + (h - 40.0) * 1.5 * r$
fi ▮

From these examples, you can see that GCL requires the programmer to be more explicit in the precondition that must be true for one of the alternatives of an **if** statement to execute. In Example 6.19, you cannot tell simply by reading the code what must be true for the false alternative to execute. You must deduce that for the false alternative to execute the boolean expression

```
d.hours <= 40.0
```

must be false, and from that fact reason that d.hours > 40.0 must be true. On the other hand, in GCL the guard tells you explicitly what must be true for the else part to execute.

Exercises

1. State whether the boolean expression

```
ODD(num1) & (num2 <= 10)
```

is true or false for each of the following sets of values for the integer variables num1 and num2.

(a) num1 = 6, num2 = 10
(b) num1 = 5, num2 = 11
(c) num1 = 5, num2 = 10

2. State whether the boolean expression

```
(num1 >5) OR (num2 <= 12)
```

is true or false for each of the following sets of values for the integer variables num1 and num2.

(a) num1 = 20, num2 = 12
(b) num1 = 7, num2 = 8
(c) num1 = 2, num2 = 13

3. Write the equivalent of the following IF tests without using the ~ operator.

(a) IF ~(num < 16) THEN
(b) IF ~((num1 < 20) OR (num2 >= 10)) THEN
(c) IF ~((num1 = 20) & (num2 > 10)) THEN

4. Predict the output of the program in Figure 6.6 for the following inputs.

(a) d.hours = 38.0, d.rate = 4.75
(b) d.hours = 50.0, d.rate = 5.00
(c) d.hours = –2.0, d.rate = 10.00

5. Predict the output of the program in Figure 6.10 for the following inputs.

(a) d.hours = 36.0, d.rate = 5.00
(b) d.hours = 48.0, d.rate = 6.00
(c) d.hours = –1.0, d.rate = 10.00

6. Predict the output of the program in Figure 6.13 for the following inputs.

(a) d.fare = 100.00, d.numFlights = 9, d.olderThan65 = false
(b) d.fare = 100.00, d.numFlights = 19, d.olderThan65 = true
(c) d.fare = 100.00, d.numFlights = 14, d.olderThan65 = true

7. Draw the flowcharts for the procedures in **(a)** Figure 6.10 and **(b)** Figure 6.13.

8. Draw the flowcharts for the following code fragments.

(a)

```
IF Condition 1 THEN
    Statement 1
ELSE
    Statement 2 ;
    Statement 3
END
```

(b)

```
IF Condition 1 THEN
    Statement 1
ELSE
    Statement 2
END;
Statement 3
```

9. Simplify the following code fragment. Assume that none of the statements change the variables in Condition 1.

```
IF Condition 1 THEN
    Statement 1 ;
    Statement 2
ELSE
    Statement 1 ;
    Statement 3
END
```

10. Rewrite the following code fragments with the correct indentation and draw their flowcharts.

(a)

```
Statement 1 ;
IF Condition 1 THEN
Statement 2
ELSE
Statement 3
END;
Statement 4 ;
Statement 5
```

(b)

```
Statement 1 ;
IF Condition 1 THEN
Statement 2
ELSE
Statement 3 ;
Statement 4
END;
Statement 5
```

11. Translate the following code fragments from CP to GCL.

(a)
```
IF d.age > 65 THEN
    rate := 0.2 * d.wages
ELSE
    rate := 0.3 * d.wages
END
```

(b)
```
IF d.xCoordinate > 1000 THEN
    d.xCoordinate := d.xCoordinate - 1000
END
```

12. Translate the following code fragments from GCL to CP.

(a)
```
if   $a \geq b \rightarrow a, b := b, a$
 []  $a < b \rightarrow$ skip
fi
```

(b)
```
if   $j > 100 \rightarrow m :=$ “large”
 []  $j \leq 100 \rightarrow m :=$ “small”
fi
```

Problems

13. A salesperson’s commission is computed as 15% of the sales that exceed $1000. Write a Component Pascal program to input a sales figure from a dialog box and output the salesperson’s commission in a dialog box message. Use an IF statement without an ELSE part.

14. In a bowling tournament, participants bowl three games and receive a consolation prize of $15 regardless of their score. Those bowlers whose three-game average exceeds 200 get an additional prize of $50. Write a program to input a bowler’s three scores from a dialog box and output his prize earnings in a dialog box message.

15. Write a Component Pascal program to input two integer values from a dialog box and output them in numeric order in a dialog box message.

16. A student gets on the dean’s list if her grade point average (GPA) is at least 3.5 (based on a scale of 4.0 for an A, 3.0 for a B, etc.). Write a program that implements a dialog box with input fields for the number of A’s, B’s, C’s, D’s, and F’s a student earned during a given semester and with output fields for her GPA and a message telling whether she made the dean’s list.

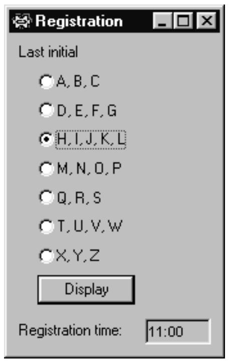
17. Design a dialog box with an integer input field for “Age” and a check box for “Dependent”. If the age field is less than 21 and the check box is checked output the message, “You qualify.”, otherwise, “You do not qualify.” in an output field in the dialog box.

18. Make the improvement described in the text to the program of Figure 6.15.

19. A person’s last initial determines her registration period, as the table in Figure 6.22(a) shows. Write a program using a CASE statement that asks a user to select the initial of her last name as shown in the dialog box of Figure 6.22(b) and output the registration period.

Last initial	**Registration period**
A, B, C	9:00
D, E, F, G	10:00
H, I, J, K, L	11:00
M, N, O, P	12:00
Q, R, S	1:00
T, U, V, W	2:00
X, Y, Z	3:00

(a) Table of registration periods.



Last initial: [✓ H, I, J, K, L]

Registration time: 11:00

(b) Dialog box.

Figure 6.22 The information for Problem 19.