Chapter 4 — Decisions

Section A: Basic Theory

Introduction

A decision asks a question that can be answered true or false, yes or no. Decisions are widely used in most programs. If a question is true, then often one or more actions are to be performed. However, if the question is false or not true, then one might have some alternative processing steps to be performed instead.

A decision can be thought of as having three parts: a test condition to be examined, one or more instructions to follow when the test condition is true, and one or more instructions to follow when the test condition is false. When considered from this point of view, the test condition itself can be used in far more C++ constructs than just a simple decision structure.

In C++, the decision structure is called an If-Then-Else.

The Components of an If-Then-Else Decision Structure

The decision structure is shown below in Figure 4.1. Notice that flow of control comes in at the top and after branching and doing one of two alternative sets of statements, control leaves out the bottom. The statements to do when the test is true are called the then-clause. The statements to do when the test is false are called the else-clause.

The If-Then-Else Syntax

The If-Then-Else basic syntax to implement the decision structure is as follows.

```
if (test condition) {
  0, 1 or more statements to do if the test condition is true
}
else {
  0, 1, or more stmts to do if the test condition is false
}
```

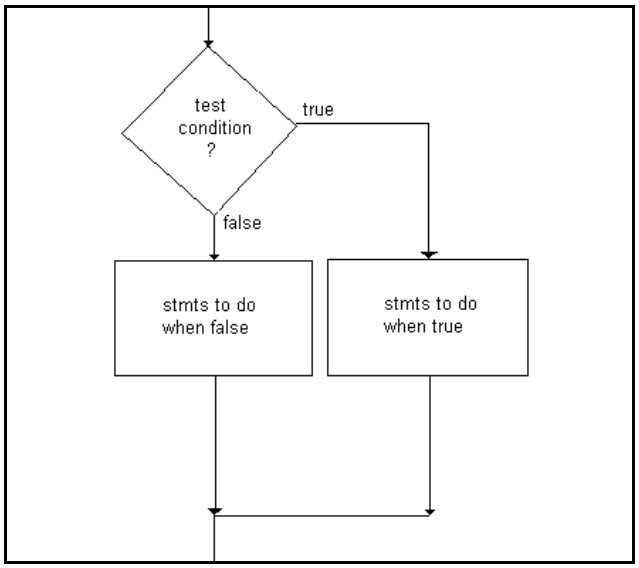


Figure 4.1 The Decision Structure

The keyword **if** begins the decision. It is followed by a test condition surrounded by parentheses. Note that there is no “then” keyword but that there is an **else** keyword.

The else-clause is strictly optional; if nothing needs to be done when the test condition is false, the else-clause does not need to be coded.

Notice that the statements to be done are surrounded by a begin-end block { }. I prefer to place the begin block { on the line that is launching that block. All statements within that block are uniformly indented. The end block } must align with the start of the line that is launching the block.

The other commonly used style looks like this.

```
if (test condition)
  {
  0, 1 or more statements to do if the test condition is true
  }
else
  {
  0, 1, or more stmts to do when the test condition is false
  }
```

In this style, the begin block { and all statements within that block and the end block } are all uniformly indented the uniform amount. Choose one style or the other and remain consistent in its use throughout the program.

The else-clause is optional. If there is nothing to do when the test condition is false, it can be omitted as shown below.

```
if (test condition) {
  0, 1 or more stmts to do when the test condition is true
}
```

Further, if there is only a single statement to do, the begin-end pair {} can be omitted.

```
if (test condition)
  a single statement when true;
else
  a single statement when false;
```

Or if there is nothing to do when the statement is false, it can be simply

```
if (test condition)
  a single statement when true;
```

Note that a single statement can be a null statement consisting of just a semicolon.

```
if (test condition)
  a single statement when true;
else
 ;
```

The Test Condition

Test conditions can be very complex and the rules, likewise. However, let's start with simple ones and add onto the complexity as we gain understanding of what they are and how they are used. A test condition asks some kind of question that can be answered true or false. In its most basic form, it parallels how we ask a question in English. For example

is the quantity less than or equal to five?
is x greater than y?
is count not equal to zero?
is sum equal to zero?

Notice in these examples that the comparison operators are less than or equal to, greater than, not equal to, and equal to. Observe that there is some quantity to the left and also to the right of each comparison operator. In English, the following would make no sense

if quantity greater than

"Greater then what" is the immediate reply. This gives us the basic format of a simple test condition:

```
operand1 comparison-operator operand2
```

In C++, there are six comparison operators. They are

```
>    greater than
>=   greater than or equal to
<    less than
<=   less than or equal to
!=   not equal to
==   equal to
```

Pay particular attention to the comparison equals operator! Notice it is a double equals sign (==); it is not a single equals sign (=). A single equal sign (=) is **always** an assignment operator in C++.

Using these operators, we can translate the above four English comparisons into C++ as follows.

```
if (quantity <= 5) {
   // do these things if true
}

if (x > y) {
   // do these things if true
}

if (count != 0) {
   // do these things if true
}

if (sum == 0) {
   // do these things if true
}

if (hoursWorked > 40) {
   // calculate overtime pay
}
```

Let's apply just this much to some programming situations. Suppose that we wanted to print a message if an employee was eligible for early retirement. That is, if their **age** was greater than or equal to 55. One could code the following to do this.

```
int age;
long employeeID;
cout << "Enter the employee id number and age\n";
cin >> employeeID >> age;
if (age >= 55) {
   cout << employeeID
        << " is eligible for early retirement\n";
}
```

Suppose that we wanted to determine whether or not a person was eligible to vote. We can input their **citizenship** status which contains a one if they are a citizen and a zero if they are not a citizen.

```
int citizenship;
cout << "Enter citizenship status: ";
cin >> citizenship;
if (citizenship == 1) {
   cout << "You are eligible to vote\n";
```

```
}
else {
   cout << "Non-citizens are not eligible to vote\n";
}
```

Ok. So far it looks fairly simple, but complexity can arise swiftly. Decisions can be nested inside each other.

Nested Decisions

There is no limit to the complexity of statements that can be contained in the then-clause or the else-clause of a decision. Hence, another decision structure could be found inside of the then-clause, for example. However, such nested decisions must be entirely contained within the then-clause. This gives us the ability to choose from among several choices, not just between two.

In the above voting example, a citizen must also be 18 or older to be eligible to vote. So realistically inside the then-clause, which is executed if the person is a citizen, we need to further test to see if the person is old enough to vote. And here programmers can get into trouble.

Consider this version in which I have manually added line numbers for reference.

```
1.   if (citizenship == 1)
2.        if (age < 18)
3.           cout << "You must be 18 to be eligible to vote\n";
4.   else
5.      cout << "Non-citizens are not eligible to vote\n";
```

Notice the nice block structure. It “looks” reasonable. However, it is not correct. Suppose that the **citizenship** is a one and the **age** is 50. What actually prints out is “Non-citizens are not eligible to vote.” Why? White space is the delimiter in C++. Thus, the nice alignment of line 4’s else with line 1’s if statement makes no difference to the compiler. Line 4’s else-clause actually is the else-clause of line 2’s if statement!

There are several ways to code the nested if statements correctly. One way is to provide the missing else-clause for line 2's **if** statement.

```
if (citizenship == 1)
     if (age < 18)
        cout << "You must be 18 to be eligible to vote\n";
     else
        ;
else
   cout << "Non-citizens are not eligible to vote\n";
```

Here the else-clause has been provided and consists of a null statement, that is, a simple semicolon.

However, the real genus of the coding error came from not using begin-end braces {} around the two clauses. If you always use the braces, you will be far less likely to code these inadvertent errors. Here is perhaps the best way to repair the coding.

```
if (citizenship == 1) {
     if (age < 18) {
        cout << "You must be 18 to be eligible to vote\n";
     }
}
else {
   cout << "Non-citizens are not eligible to vote\n";
}
```

By using the begin-end braces on the then-clause of the citizenship test, the compiler knows for certain that there can be no else-clause on the age test because the age decision must be contained within the then-clause of the citizenship test.

Let’s look at an even more complex decision structure. Suppose that our company is asked to check up on its hiring practices. After inputting the information on an employee, we are to display a message if that person is over age 50 or is physically challenged or if their race is not Caucasian. The input fields consist of age, disability (1 if so), and race (1 if white). We can code the decisions as follows.

```
if (age > 50) {
   cout << "Over 50 ";
}
else if (disability == 1) {
   cout << "Disabled ";
}
else if (race != 1) {
   cout << "Non-white ";
}
```

Notice how the else-clauses use a single statement which is itself another If-Then-Else statement. It could also be coded this way.

```
if (age > 50) {
  cout << "Over 50 ";
}
else {
   if (disability == 1) {
     cout << "Disabled ";
   }
   else {
      if (race != 1) {
          cout << "Non-white ";
      }
   }
}
```

It could also be coded this way

```
if (age > 50)
```

```
      cout << "Over 50 ";
   else
      if (disability == 1)
        cout << "Disabled ";
      else
         if (race != 1)
            cout << "Non-white ";
```

Probably the first way is the easiest to read. Okay. What does the program display for output if one enters a 55-year-old African-American who has a limp? The age test is checked first and out comes the fact that this employee is over 50. There is no mention of the other aspects. Suppose that we restate the problem to display all of the possible aspects an employee might have. How would the coding change? Notice that the reason the second test was never performed with the current employee is that all the other tests began with an else, meaning only check further if the age was not more than 50. If we just remove the else's, we are left with three independent decisions that are not nested in any way.

```
if (age > 50) {
   cout << "Over 50 ";
}
if (disability == 1) {
   cout << "Disabled ";
}
if (race != 1) {
   cout << "Non-white ";
}
```

Now the output would be "Over 50 Disabled Non-white."

When programming decisions, one must be very careful to duplicate precisely the problem's specifications.

Suppose that we are running a dating service. A client wishes to see if a specific person would be a possible match for them. The client wishes to see if this candidate is a single male between the ages of 20 and 25. Assume that we have input the **age**, **maritalStatus** (0 for single), and **sex** (1 for male). In this example, to be a possible match, the candidate must satisfy all four tests, single, male, age greater than or equal to 20 and age less than or equal to 25. Notice that you cannot write

```
if (20 <= age => 25){
```

It requires two separate test conditions. Notice also that all four of these test conditions must be true for us to display the message. Here is how this might be coded.

```
if (age >= 20)
   if (age <= 25)
      if (maritalStatus == 0)
         if (sex == 1)
            cout << "Is a potential match\n";
```

It could also be written this way.

```
if (age >= 20) {
   if (age <= 25) {
      if (maritalStatus == 0) {
         if (sex == 1) {
            cout << "Is a potential match\n";
         }
      }
   }
}
```

It could also be written this way.

```
if (age >= 20)
   {
   if (age <= 25)
      {
      if (maritalStatus == 0)
         {
         if (sex == 1)
            {
            cout << "Is a potential match\n";
            }
         }
      }
   }
```

Here is another example. Suppose that the month has been input. Display the message "Summer Vacation" if the month is June, July or August. For any other months, print "School in session." Please note that you cannot code

```
if (month == 6, 7, 8) // does not compile
```

Each value must be a complete test condition. If the month is 6 or if the month is 7 or if the month is 8, then print the message.

```
if (month == 6)
   cout << "Summer Vacation\n";
else if (month == 7)
   cout << "Summer Vacation\n";
else if (month == 8)
   cout << "Summer Vacation\n";
else
   cout << "School in session\n";
```

Notice that an else verb connects each decision after the first one.

Now suppose that we did not have to output the last message if school was in session. One might be tempted to code the following.

```
if (month == 6)
   cout << "Summer Vacation\n";
if (month == 7)
   cout << "Summer Vacation\n";
if (month == 8)
```

```
        cout << "Summer Vacation\n";
```

Yes, it still produces the correct answer. But this raises a serious efficiency concern. If the **month** contains a six, then after printing the message, control passes to the next decision. But since it contained a six, it cannot under any circumstances also contain a seven or an eight! Yet, the program wastes time retesting the **month** for a seven and then for an eight. This kind of programming is wasteful of computer resources and is generally frowned upon in the industry. It shows a distinct lack of thought on the part of the programmer. Don't do it.

Compound Test Conditions

The previous test conditions have become rather lengthy and a bit unwieldy. There is a much better alternative to so much coding. A test condition can be a compound one. That is, two or more tests can be joined together to form a larger test using either the **AND** or **OR** relational operators. First, let's define AND and OR logic.

AND logic says that both tests must be true to get a true result. This is often expressed using boolean (two-valued) logic.

```
    true      true       false      false
AND true  AND false  AND true   AND false
    -----     -----      -----      -----
    true      false      false      false
```

OR logic says that if either one or both of the tests are true, the result is true. Expressed in boolean logic, OR logic appears as follows.

```
    true      true       false      false
 OR true   OR false   OR true    OR false
    -----     -----      -----      -----
    true      true       true       false
```

In C++ these two operators are **&&** for AND and **||** for OR. To either side of these operators must be a test condition. I often refer to these two operators as the "joiner ops" since they are used to join two tests together. Figuratively, if we code

```
if (test1 && test2)
```

then this is saying that if test1 **and** test2 are both true, then execute the then-clause. Again, figuratively if we code

```
if (test1 || test2)
```

then this is saying that if either test1 **or** test2 is true, then execute the then-clause.

We can greatly simplify the previous examples using these compound joiner operators. In the dating service example, all the tests had to be true before the program printed the potential match message. This means AND logic is used and the joiner would be **&&**. Rewriting those four decisions into one greatly simplifies that decision.

```
if (age >= 20 && age <= 25 && maritalStatus == 0 && sex == 1)
 cout << "Is a potential match\n";
```

The summer vacation decision is an example of OR logic, since if either of the three month tests is true, it is a summer month. It can be simplified as follows using the || operator.

```
if (month == 6 || month == 7 || month == 8)
   cout << "Summer Vacation\n";
else
   cout << "School in session\n";
```

These two operators are actually even more efficient. Take the AND operator **&&** for example. Suppose that the age of the person in the above dating service problem was 19. When the very first test condition is executed, the age is not greater than or equal to 20. Thus, a false results. Since the **&&** operator is joining all of these other test conditions, C++ immediately knows the final outcome of the compound test, false. Remember, AND logic says that they all have to be true to get a true result. And that is exactly how C++ behaves. As soon as one of the joined tests yields a false result, the compiler stops doing the remaining tests and immediately goes to the else-clause if there is one. The remaining tests are not even executed! C++ is being as efficient as it can with compound test conditions. Programmers often take advantage of just this behavior, terminating remaining tests joined with the **&&** operator when a false result is encountered. We will do just that in Chapter 9.

The same efficiency applies to test conditions that are joined with OR **||** operators. Consider the summer vacation tests. If the month is indeed six, then the result of the first test is true. Since the remaining tests are joined with **||** operators, the compiler immediately knows the final result, true, and does not bother to perform the remaining tests, jumping immediately into the then-clause.

Thus, when we join our related decisions using **AND** and **OR** operators to form longer tests, we gain a measure of efficiency from the language itself.

The **AND** operator has a higher precedence than does the **OR** operator and both are lower than the six relational operators. In the dating service example, we could have used parentheses as shown below, but because the **&&** is at a lower precedence, they are not needed.

```
if ( (age >= 20) && (age <= 25) &&
     (maritalStatus == 0) && (sex == 1))
 cout << "Is a potential match\n";
```

Both **&&** and **||** operators can be used in the same decision. If so, the **&&** is done before the **||** operation.

```
if (a > 5 && b > 5 || c > 5)
```

This is the same as if we had coded

```
if ( (a > 5 && b > 5) || c > 5)
```

Sometimes, parentheses can aid readability of a program. It does not hurt to use them in that manner.

Also note that the six relational operators have a lower precedence than all of the math operators. Thus, if we coded the following

```
if (a + b > c + d)
```

then this would group as if we had used parentheses:

```
if ((a + b) > (c + d))
```

However, I tend to use the parentheses anyway because it aids program readability. Without them, the reader must know that the relational operator > is of lower precedence.

The Logical Not Operator — !

The last of the logical operators is the not (**!**) operator. This operator reverses the condition to it's right. Suppose that one coded the following.

```
if ( ! (x > y) )
```

The **!** reverses the result. If **x** contains 10 and **y** contains 5, then the test **x > y** is true, but the **!** reverses it; the true becomes false and the else branch is taken. However, if **x** equals **y** or is actually less than **y**, then the test **x > y** is false; the **!** reverses that to true and the then-clause is executed.

Confusing? Likely so. I have more than 35 years experience in the programming world. One thing that I have found to be uniformly true among all programmers is confusion over not-logic. While no one has any trouble with test conditions like **x != y**, there is uniform non-comprehension about not-logical conditions, such as the one above. In fact, the chances of mis-coding a complex not-logical expression are exponential! My advice has always been "Reverse it; say the test in the positive and adjust the clauses appropriately."

Here is another example to illustrate what I mean. In the dating service match test condition, the age is to be between 20 through and including 25. That is, an age that is less than 20 or an age that is more than 25 are not candidates for a match. Thus, one could test for those using

```
age < 20 || age > 25
```

But if these were true, then this person is not a match and we are looking for a match, so not logic would reverse it.

```
if ( !((age < 20) || (age > 25)) &&
     (maritalStatus == 0) && (sex == 1))
 cout << "Is a potential match\n";
```

This is much more difficult to read for the average programmer. Unless your mathematical background is well attuned to these kinds of logical expressions, it is much better to reverse the not logic and say it in the positive as in the following.

```
if ( (age >= 20) && (age <= 25) &&
     (maritalStatus == 0) && (sex == 1))
 cout << "Is a potential match\n";
```

There are a few times where not logic improves the coding by making it tighter, shorter and more compact, mostly in the area of controlling the iterative or looping process (next

chapter). I use not logic sparingly throughout this text, preferring to always try to say it in the positive manner.

Data Type and Value of Relational Expressions — The bool Data Type

In C++, the result of any test condition or relational expression is always an **int** data type whose value is either 0 for false or 1 for true.

When the test condition evaluation is completed by the compiler, the result is an **int** whose value is either 0 or 1. Assume that integer variable **x** contains 10 and integer variable **y** contains 5. In the following **if** statement

```
if (x > y)
```

the test condition **x** > **y** evaluates to true or an integer 1. The compiler then sees just

```
if (1)
```

and so does the then-clause. If however, we reverse the contents of variables **x** and **y**, then the test results in a false and the compiler sees just

```
if (0)
```

and takes the else-clause if present.

This means that one could define an integer variable to hold the result of a relational expression. Consider the following code.

```
int result = x > y;
```

At first, it may look a bit bizarre. To the right of the assignment operator is a test condition. After the test is complete, the result is either 0 or 1 and that **int** value is what is being assigned to the variable **result**. Another way of looking at the variable **result** is that it is holding a true/false value.

The bool Data Type

However, there is a far better data type to use if only a true/false value is desired. This is the new data type called **bool** which represents boolean data or two-valued logic. A variable of type **bool** can have only two possible values, **true** and **false**. Some examples of variables that can effectively utilize this data type include the following.

```
bool isWindowVisible; // true if this window is visible
bool isMoving;        // true if this object is in motion
bool isAvailable;     // true if available for work
bool isFoodItem;      // true if this item is classified as
                      //      food for tax purposes
```

Consider the readability of this section of coding that deals with moving objects.

```
bool isMoving;
...
if (mph > 0)
```

```
    isMoving = true;
  else
    isMoving = false;
  ...
  if (isMoving) {
   ...
  }
```

If the variable **mph** contains say 45 miles an hour, then the test is true and a true value is assigned to **isMoving**. Notice that the test condition could also have been written

```
  if (isMoving == true) {
   ...
  }
```

But since a **bool** already contains the needed 1 or 0 value (**true** or **false**), it is not needed.

Using **bool**s can add a measure of readability to a program. Consider using a **bool** whenever the variable can be expressed in a true/false manner.

The compiler can always convert an integer relational expression result into a **bool** data type. The above coding can be rewritten even simpler as follows.

```
  isMoving = mph > 0;
  ...
  if (isMoving) {
   ...
  }
```

This leads us to the two vitally important shortcut test conditions that are widespread in C++ coding. To summarize, if we code

```
  if (x < y)
```

then this results in an integer value 1 or 0 which is then evaluated to see if the then-clause or else-clause is taken

```
  if (1)
```

or

```
  if (0)
```

Notice that the then-clause is executed if the test result is not 0. The else-clause is taken if the result is equal to 0. This is commonly extended by coding these two shortcuts

```
  if (x)
```

or

```
  if (!x)
```

where **x** is a variable or expression.

When the compiler encounters

```
  if (x)
```

it checks the contents of variable **x**. If **x**'s contents are non-zero, the then-clause is executed. Similarly, when the compiler encounters

```
  if (!x)
```

it checks the contents of variable **x**. If **x**'s contents are zero, then the then-clause is taken. Thus,

```
if (x)
```

is a shortcut way of saying

```
if (x != 0)
```

and

```
if (!x)
```

is a shortcut way of saying

```
if (x == 0)
```

I commonly keep track of these two shortcuts by using this scheme. If something does not exist, it is zero. So **if (!apples)** means not apples means no apples or that apples is zero. And **if (apples)** means if apples exist and apples exist if apples is not zero. However, you choose to remember these two shortcuts, make sure you understand them for their use is widespread in advanced C++ programming.

The Most Common Test Condition Blunder Explained

At long last, we can finally understand the most common error made by programmers when coding test conditions. And that error is shown here.

```
if (quantity = 0)
```

or

```
if (x = y)
```

The error is coding an assignment operator instead of the conditional equality operator ==. This is not a syntax error and does compile and execute, but the results are disastrous. Why? It is first and foremost an assignment operator. When either of the above two if instructions are executed, the first action is to replace the value in **quantity** with a 0 and to replace the contents of variable **x** with the contents of variable **y**. This is therefore destructive of the contents of **quantity** and **x**! Secondly, once the value has been copied, the compiler is left with just evaluating

```
if (quantity)
```

and

```
if (x)
```

But we now know what that actually becomes. If the newly updated **quantity** is not zero, take the then-clause. If the newly updated **x** is not zero, take the then-clause.

This coding action is exceedingly rarely used. Thus, many compilers actually issue a warning message along the lines of "assignment in a test condition!"

Always be extra careful to make sure you use the equality relational operator == and not the assignment operator = when making your test conditions.

The Conditional Expression

The conditional expression operators **? :** provide a shortcut to the normal If-Then-Else coding. Let's calculate the car insurance premium for a customer. The rates are based on the age of the insured. The following calculates the insured's premium using If-Then-Else logic.

```
double premium;
if (age > 55)
  premium = 100.00;
else
  premium = 250.00;
```

Notice in both clauses, something is being assigned to the same variable, **premium**, in this case. That is the key that the conditional expression can be used. It would be coded like this

```
double premium = age > 55 ? 100.00 : 250.00;
```

The syntax of the conditional expression is

```
test condition ? true expression : false expression
```

The test condition is the same test condition we have been discussing this whole chapter. It can be simple or compound. It is followed by a ? After the ? comes the then or true expression. The expression can be as simple as a constant as in this case or it can be a variable or an expression that results in a value to be used. After the true expression comes a : to separate the true and false expressions and the false expression follows.

Here is a more complex version. Suppose that younger drivers pay a higher rate. We now have the following.

```
double premium;
if (age > 55)
  premium = 100.00;
else if (age < 21)
  premium = 1000;
else
  premium = 250.00;
```

This can be rewritten as follows

```
double premium = age > 55 ? 100 : age < 20 ? 1000 : 250;
```

Here the false portion of the first conditional expression is another entire conditional expression! But more importantly, we have reduced seven lines of coding into one line.

The conditional expression is not limited to assignments, though it is commonly used in such circumstances. Suppose that we need to make a very fancy formatted line indicating the higher temperature for the day. The line is to be shown on the 10 o'clock weather. Assume that we have already calculated the morning and evening temperatures and need now to display the larger of the two temperatures as the higher temperature. We could do the following.

```
if (am_temp > pm_temp)
  cout << ...fancy formatting omitted << am_temp << endl;
else
  cout << ...fancy formatting omitted << pm_temp << endl;
```

While this works well, as you have undoubtedly discovered at this point, making output look good requires a lot of trial and error, fiddle, fiddle, to get it to look good. Here in this example, we have precisely the same fancy formatting to do twice! As you tweak the first output, you must remember to do the same exact things to the second output instruction. I have enough trouble getting it right once. The conditional expression comes to our rescue. This can be rewritten as

```
cout << ...fancy formatting omitted
     << am_temp > pm_temp ? am_temp : pm_temp
     << endl;
```

This then displays the larger of the two temperatures.

The Precedence of Operators

Table 4.1 shows the precedence of operators from highest at the top to the lowest at the bottom. Each row is at a different level. A function call must always be done first so that the value the function returns is available for the rest of the expression's evaluation. Assignments are always last. Of course, parentheses can be used to override the normal precedence.

Notice that the postfix operator (after inc or dec) and prefix operator (before inc and dec) have high precedence so that their use can be detected early and properly applied after the current value is used.

The unary - is used in instructions such as

```
x = - y;
```

The address operator & returns the memory location of the item that follows it. So if we coded

```
&x
```

this returns where in memory variable **x** is located. We deal with addresses in a later chapter.

Table 4.1 The Precedence of Operators

Operator	Name	Associates
functionName (...)	function call	left to right
++ and --	postfix increment and decrement operators	left to right
++ and --	prefix increment and decrement operators	right to left
-, +, !, &	unary -, unary +, logical not, and address of operators	right to left
(datatype)	typecast	right to left

Operator	Name	Associates
*, / and %	multiply, divide and remainder operators	left to right
+ and -	add and subtract operators	left to right
>, >=, <, <=	greater than, greater than or equal, less than, less than or equal to operators	left to right
== and !=	equal to and not equal to operators	left to right
&&	logical and	left to right
||	logical or	left to right
? :	conditional expression	right to left
=, +=, -=, *=, /=, %=	assignment operators	right to left

Testing of Real Numbers

There remains one additional test condition situation that must be understood. This applies only to floating point or real numbers. Recall that a floating point number is only an approximation of a specific real number, as close as the computer can get in the finite number of binary decimal bits. Further, when calculations are done on these floating point numbers, small roundoff errors and precision effects begin appearing.

For example, let's take a variable **x** of **float** data type. Suppose that it is initialized to 4.0. The computer stores this number as close as it can get to 4.0. It might be 3.99999 or it might be 4.00001. Now assume that we do the following to **x**

```
x = x - y
```

where **y** contains 3.99998. At this point, variable **x** contains 0.00001 or 0.00003, depending upon the above values. What happens if we do the following test condition?

```
if (x == 0)
```

Clearly, the else-clause is taken because **x** is not zero. But it sure is close to zero! The question is "is **x** sufficiently close to zero to be actually considered zero?" Or wilder still, suppose **x** was a **double** that contained 0.00000000000000000123456789012345. A **double** has fifteen digits of accuracy and that is what is stored here — .123456789012345E-15. Is this version of **x** zero? Nope.

When testing floating point numbers, to avoid this kind of error, always test in such a manner to see if it is sufficiently close to the desired value. Use the floating point absolute value function.

```
if (fabs (x) <= .000001)
```

This takes the absolute value of **x** and compares it to the desired degree of closeness. If you have no idea how close is close enough, try one part in a million or .000001.

Similarly when comparing two floating point values for equality, always compare the absolute value of their difference to the desired degree of closeness. Instead of coding

```
if (x == y)
```

code

```
if (fabs (x - y) <= .000001)
```

How close is "close enough" depends on the problem at hand. Suppose that **x** represents the cubic yards of concrete to place into a concrete truck to deliver to a construction site. Probably .1 is highly accurate enough!

Section B: Computer Science Example

Cs04a — Compute the Total Bill By Finding the Sales Tax Rate

Acme Company sells products in two states. Typically, state codes of 13 for Illinois and 14 for Iowa are used to determine the tax rates. Assume that the Illinois tax rate is 7.5% and the Iowa rate is 8%. Additionally, if 10 or more than of the same item are purchased, a discount of 4% is given on the total cost of those items. If the total sale is $100.00 or more, shipping costs are free. Otherwise the customer pays shipping which is $4.00 or .5% of the total order before taxes, whichever is larger. Write a program that inputs one order consisting of the customer number (up to six digits long), the state code number, the item number of the product ordered, the quantity ordered and the unit cost. Print out a nice billing form showing the order details and final total cost to the customer.

The design begins as usual by identifying the input fields. Here we need **custNumber**, **stateCode**, **itemNumber**, **quantity**, and **cost**. Draw a set of main storage boxes for these and label them with their chosen names. Figure 4.2 shows the complete main storage diagram.

Now using these names, write out the steps to solve this problem.

Prompt and input **custNumber**, **stateCode**, **itemNumber**, **quantity**, and **cost**

The first calculation is to find the total cost of the quantity purchased. Then we can apply discounts. Let's call this one **subTotal**; add another box in the main storage diagram for it and write

subTotal = **quantity** * **cost**;

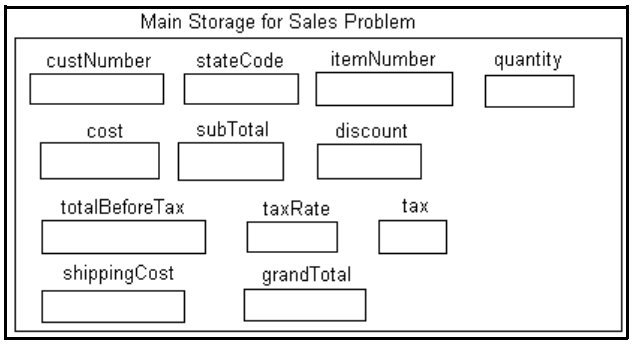


Figure 4.2 Main Storage for Sales Tax Problem

Next, check the **quantity** ordered and see if there is a discount to be applied.
if the **quantity** is greater than or equal to 10, then do the following
    discount = **subTotal** * .04;
otherwise
    discount is 0;
Next, figure the total before tax, calling it **totalBeforeTax**
totalBeforeTax is **subTotal** minus **discount**

To figure the tax, we need to get the rate. Let's call it **taxRate**; make a box for it and **tax**. Then calculate them by
if **stateCode** is equal to 13 then
    taxRate = . 75
else check if **stateCode** is equal to 14 if so then
    taxRate = . 8
tax = **totalBeforeTax** * **taxRate**

To get the shipping costs, we need a field to hold it, say **shippingCost** and it is calculated as follows
if **totalBeforeTax** is greater than or equal to 100 then
    shippingCost is 0
otherwise
    shippingCost = **totalBeforeTax** * .005
    but if **shippingCost** < 4.00 then
        shippingCost = 4;

Finally, the grand total due, say called **grandTotal**, is given by the sum of the following partial totals.
grandTotal = **totalBeforetax** + **shippingCost** + **tax**
now display all these results nicely formatted

One should thoroughly desk check the design. Make up various input sets of data so that

all possible situations can occur and be verified.

```
     customer  state  item     qty    cost
     number    code   number
      12345      13    1111     5     10.00 // no discounts
      12345      13    1111    10     10.00 // only 4%
      12345      13    1111    15     10.00 // 4% & free shipping
     123456      14    1111     5     10.00 // other state rate
```

Here are the completed program and the output from the above four test executions.

```
+)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Cs04a: Customer Order Program                                             *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  1 /*************************************************************/*
*  2 /*                                                           */*
*  3 /* Cs04a: Customer Order                                     */*
*  4 /*                                                           */*
*  5 /*************************************************************/*
*  6                                                               *
*  7 #include <iostream.h>                                        *
*  8 #include <iomanip.h>                                        *
*  9                                                               *
* 10 int main () {                                                *
* 11                                                              *
* 12  // input variables                                         *
* 13  long   custNumber;  // customer name                       *
* 14  int    stateCode;   // state code 13 or 14                 *
* 15  long   itemNumber;  // item number ordered                 *
* 16  int    quantity;    // quantity ordered                    *
* 17  double cost;        // cost of one item                    *
* 18                                                              *
* 19  // prompt and input a set of data                          *
* 20  cout << "Enter Customer Id number: ";                      *
* 21  cin >> custNumber;                                         *
* 22  cout << "Enter state code:        ";                      *
* 23  cin >> stateCode;                                          *
* 24  cout << "Enter Item number:       ";                      *
* 25  cin >> itemNumber;                                         *
* 26  cout << "Enter quantity ordered:  ";                      *
* 27  cin >> quantity;                                           *
* 28  cout << "Enter cost of one item:  ";                      *
* 29  cin >> cost;                                               *
* 30                                                              *
* 31  // the calculation fields needed                          *
* 32  double subTotal;       // basic cost of these items       *
* 33  double discount = 0;   // 4% discount if quantity >= 10    *
* 34  double totalBeforeTax; // total ordered with discount applied *
* 35  double taxRate;        // tax rate based on state code    *
* 36  double tax;            // total tax on totalBeforeTax     *
* 37  double shippingCost = 0; // shipping free if totalBeforetax>=100*
* 38  double grandTotal;     // total due from customer         *
* 39                                                              *
```

```
* 40  // the calculations section                                        *
* 41  subTotal = quantity * cost; // figure basic cost                  *
* 42                                                                    *
* 43  if (quantity >= 10)         // apply 4% if quantity large enough*
* 44   discount = subTotal * .04;                                       *
* 45  totalBeforeTax = subTotal - discount; // total before taxes      *
* 46                                                                    *
* 47  if (stateCode == 13)        // find the right tax rate to use    *
* 48   taxRate = .075;            // state Illinois rate               *
* 49  else if (stateCode == 14)                                         *
* 50   taxRate = .08;             // state Iowa rate                   *
* 51  else {                      // oops, not a valid state code      *
* 52   cout << "Invalid state code. It was " << stateCode << endl;     *
* 53   return 1;                                                       *
* 54  }                                                                 *
* 55                                                                    *
* 56  tax = totalBeforeTax * taxRate; // calc the tax owed              *
* 57                                                                    *
* 58  if (totalBeforeTax < 100) {    // need to figure shipping costs *
* 59   shippingCost = totalBeforeTax * .005;                            *
* 60   if (shippingCost < 4.00)      // if it is less than minimum amt*
* 61    shippingCost = 4.00;         // reset shipping to minimum amt *
* 62  }                                                                 *
* 63                                                                    *
* 64  grandTotal = totalBeforeTax + shippingCost + tax;                 *
* 65                                                                    *
* 66  // setup floating point format for output of dollars             *
* 67  cout.setf (ios::fixed, ios::floatfield);                          *
* 68  cout.setf (ios::showpoint);                                       *
* 69  cout << setprecision (2);                                         *
* 70                                                                    *
* 71  // display the results section                                    *
* 72  cout << endl << endl << "Acme Customer Order Form\n";            *
* 73  cout << "Customer Number: " << custNumber << " in State: "       *
* 74       << stateCode << endl;                                        *
* 75  cout << "Item Number   Quantity   Cost       Total\n";           *
* 76  cout << setw (8) << itemNumber << setw (11) << quantity          *
* 77       << setw(11) << cost << setw (11) << subTotal <<endl <<endl;*
* 78  cout << "Total after discount:"<<setw(20)<<totalBeforeTax<<endl;*
* 79  cout << "Tax:                 "<<setw(20)<<tax << endl;         *
* 80  cout << "Shipping costs:      "<<setw(20)<<shippingCost << endl;*
* 81  cout << "Grand Total Due:     "<<setw(20)<<grandTotal << endl;  *
* 82                                                                    *
* 83  return 0;                                                         *
* 84                                                                    *
.)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
+)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Output from 4 test runs of Cs04a: Customer Order Program             *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  1 Test Run # 1 Results                                              *
*  2                                                                   *
*  3 Enter Customer Id number: 12345                                   *
```

```
*  4 Enter state code:         13                                  *
*  5 Enter Item number:        1111                                *
*  6 Enter quantity ordered:   5                                   *
*  7 Enter cost of one item:   10                                  *
*  8                                                               *
*  9                                                               *
* 10 Acme Customer Order Form                                      *
* 11 Customer Number: 12345 in State: 13                           *
* 12 Item Number   Quantity   Cost       Total                     *
* 13     1111          5      10.00      50.00                     *
* 14                                                               *
* 15 Total after discount:               50.00                     *
* 16 Tax:                                 3.75                     *
* 17 Shipping costs:                      4.00                     *
* 18 Grand Total Due:                    57.75                     *
* 19                                                               *
* 20 ========================================                      *
* 21 Test Run # 2 Results                                          *
* 22                                                               *
* 23 Enter Customer Id number: 12345                               *
* 24 Enter state code:         13                                  *
* 25 Enter Item number:        1111                                *
* 26 Enter quantity ordered:   10                                  *
* 27 Enter cost of one item:   10                                  *
* 28                                                               *
* 29                                                               *
* 30 Acme Customer Order Form                                      *
* 31 Customer Number: 12345 in State: 13                           *
* 32 Item Number   Quantity   Cost       Total                     *
* 33     1111         10      10.00     100.00                     *
* 34                                                               *
* 35 Total after discount:               96.00                     *
* 36 Tax:                                 7.20                     *
* 37 Shipping costs:                      4.00                     *
* 38 Grand Total Due:                   107.20                     *
* 39                                                               *
* 40 ========================================                      *
* 41 Test Run # 3 Results                                          *
* 42                                                               *
* 43 Enter Customer Id number: 12345                               *
* 44 Enter state code:         13                                  *
* 45 Enter Item number:        1111                                *
* 46 Enter quantity ordered:   15                                  *
* 47 Enter cost of one item:   10                                  *
* 48                                                               *
* 49                                                               *
* 50 Acme Customer Order Form                                      *
* 51 Customer Number: 12345 in State: 13                           *
* 52 Item Number   Quantity   Cost       Total                     *
* 53     1111         15      10.00     150.00                     *
* 54                                                               *
* 55 Total after discount:              144.00                     *
```

```
* 56 Tax:                                10.80                          *
* 57 Shipping costs:                      0.00                          *
* 58 Grand Total Due:                   154.80                          *
* 59                                                                    *
* 60 ======================================                             *
* 61 Test Run # 4 Results                                               *
* 62                                                                    *
* 63 Enter Customer Id number: 123456                                   *
* 64 Enter state code:         14                                       *
* 65 Enter Item number:        1111                                     *
* 66 Enter quantity ordered:   5                                        *
* 67 Enter cost of one item:   10                                       *
* 68                                                                    *
* 69                                                                    *
* 70 Acme Customer Order Form                                           *
* 71 Customer Number: 123456 in State: 14                               *
* 72 Item Number   Quantity   Cost       Total                          *
* 73     1111          5      10.00      50.00                          *
* 74                                                                    *
* 75 Total after discount:               50.00                          *
* 76 Tax:                                 4.00                          *
* 77 Shipping costs:                      4.00                          *
* 78 Grand Total Due:                    58.00                          *
.)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

Section C: An Engineering Example

Engr04a — Quadratic Root Solver

A major usage of decisions is to avoid doing calculations when one or more variables are out of range for that calculation. For example, an attempt to divide by zero causes a program crash. Passing values out of range to the arcsine function cause the **asin()** function to crash the program. Commonly, decisions protect programs from such attempts. This example explores these uses.

Write a program that displays the roots of the quadratic equation, $\mathbf{ax^2 + bx + c}$, given any user inputted values for **a**, **b** and **c**.

Analyzing the problem, the equation we need to solve is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

But the complicating factor is that the user might enter values such that imaginary roots occur or even division by zero if **a** is zero. To make a totally general program, we must handle all the possibilities. The first consideration is “Is the value the user entered for the **a** term 0?” If so, there is no solution. Next if the discriminant, $\mathbf{b^2–4ac}$, is negative, then there are two imaginary

roots given by

$$-\frac{b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a} i \ldots where \ldots i = \sqrt{-1}$$

Further, if $\mathbf{b^2 - 4ac}$ is 0, then there are two identical real roots.

Designing our solution first, we must make main storage variables for the input values. Let's call them **a**, **b** and **c**. Since the discriminant, $\mathbf{b^2 - 4ac}$, must be evaluated, let's also make a variable to hold it, say **desc**. Finally, the result variables might be **root1** and **root2**. But in the case of the imaginary roots, there are going to be an imaginary part, so let's also define **iroot** to hold the imaginary portion. Figure 4.3 shows the main storage diagram.

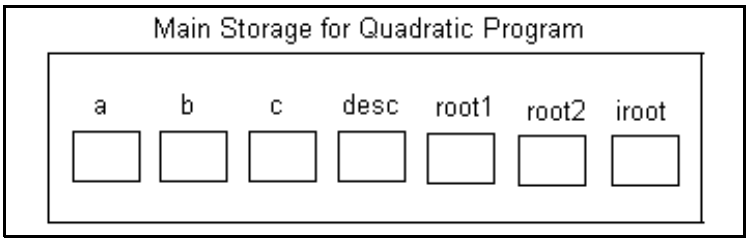


Figure 4.3 Main Storage for Quadratic Root Program

Now write out the sequence of operations needed to solve this problem using our variable names. We have

prompt and input **a**, **b** and **c**
if **a** is 0 then display no solution
otherwise to the following
    calculate **desc** = $\mathbf{b^2 - 4ac}$
    if **desc** is 0 then do the following
        find **root1** = **–b/(2a)**
        display two roots at **root1**
    otherwise if **desc** is negative then do the following
        root1 = **–b/(2a)**
        iroot = **sqrt** (|**desc**|) / (2**a**)
        display one imaginary root as **root1** + **iroot** * i
        display the other imag root as **root1** – **iroot** * i
    otherwise
        root1 = (–b+sqrt (desc))/(2a)
        root2 =(–b-sqrt (desc))/(2a)
        display **root1** and **root2**
    end otherwise
end otherwise

Next, make up some test values to thoroughly check out the program. For example, we might use these sets

0, 1, 2 - for no solution
3, 4, 5 - for imaginary roots
2, 8, 6 - for two real roots
4, 4, 1 - for multiple roots

Here are the program and the test runs.

```
+)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Engr04a - Quadratic Roots Solver                                          *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  1 /*************************************************************/*
*  2 /*                                                           */*
*  3 /* Engr04a: Quadratic Equation Roots                         */*
*  4 /*                                                           */*
*  5 /*************************************************************/*
*  6                                                                 *
*  7 #include <iostream.h>                                          *
*  8 #include <iomanip.h>                                          *
*  9 #include <math.h>                                             *
* 10                                                                *
* 11 int main () {                                                  *
* 12                                                                *
* 13  // input variables                                           *
* 14  double a;                                                     *
* 15  double b;                                                     *
* 16  double c;                                                     *
* 17                                                                *
* 18  // result variables                                          *
* 19  double desc;                                                  *
* 20  double root1;                                                 *
* 21  double root2;                                                 *
* 22  double iroot;                                                 *
* 23                                                                *
* 24  // prompt and input the user's coefficients for a, b and c    *
* 25  cout << "Quadratic Equation Root Solver Program\n\n";         *
* 26  cout<<"Enter the quadratic equation's coefficients a, b and c\n"*
* 27      << "separated by a blank\n";                              *
* 28  cin >> a >> b >> c;                                           *
* 29                                                                *
* 30  // setup floating point format for output of roots           *
* 31  cout.setf (ios::fixed, ios::floatfield);                      *
* 32  cout.setf (ios::showpoint);                                   *
* 33  cout << setprecision (4) << endl;                             *
* 34                                                                *
* 35  // check for division by 0 or not a quadratic case           *
* 36  if (fabs (a) < .000001) {                                     *
* 37   cout <<"Since a is zero, there is no solution-not quadratic\n";*
* 38  }                                                             *
* 39  // here it is a quadratic equation, sort out roots           *
```

```
* 40  else {                                                              *
* 41   desc = b * b - 4 * a * c;                                          *
* 42   // is desc basically 0 indicating multiple roots at one value?     *
* 43   if (fabs (desc) <= .000001) {                                       *
* 44    root1 = - b / (2 * a);                                            *
* 45    cout << "Multiple roots at " << setw (12) << root1 << endl;       *
* 46   }                                                                  *
* 47   // is the desc positive indicating two real roots                  *
* 48   else if (desc > 0) {                                               *
* 49    root1 = (-b + sqrt (desc)) / (2 * a);                             *
* 50    root2 = (-b - sqrt (desc)) / (2 * a);                             *
* 51    cout << "Two real roots at: " << setw (12) << root1 << endl;      *
* 52    cout << "                   " << setw (12) << root2 << endl;      *
* 53   }                                                                  *
* 54   // desc is negative indicating two imaginary roots                 *
* 55   else {                                                            *
* 56    desc = fabs (desc);                                               *
* 57    root1 = -b / (2 * a);                                             *
* 58    iroot = sqrt (desc) / (2 * a);                                    *
* 59    cout << "Two imaginary roots at :" << setw (12) << root1          *
* 60         << " + i * " << setw (12) << iroot << endl;                  *
* 61    cout << "                         " << setw (12) << root1          *
* 62         << " - i * " << setw (12) << iroot << endl;                  *
* 63   }                                                                  *
* 64  }                                                                   *
* 65                                                                      *
* 66  return 0;                                                           *
* 67                                                                      *
.)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
+)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Output from Four Test Runs of Engr04a - Quadratic Roots Solver          *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  1 Results of test tun #1                                               *
*  2                                                                      *
*  3 Quadratic Equation Root Solver Program                               *
*  4                                                                      *
*  5 Enter the quadratic equation's coefficients a, b and c               *
*  6 separated by a blank                                                 *
*  7 0 1 2                                                                *
*  8                                                                      *
*  9 Since a is zero, there is no solution - not a quadratic              *
* 10                                                                      *
* 11 Quadratic Equation Root Solver Program                               *
* 12                                                                      *
* 13 ======================================================              *
* 14 Results of test tun #2                                               *
* 15                                                                      *
* 16 Enter the quadratic equation's coefficients a, b and c               *
* 17 separated by a blank                                                 *
* 18 3 4 5                                                                *
* 19                                                                      *
* 20 Two imaginary roots at :     -0.6667 + i *       1.1055              *
```

```
*                                 -0.6667 - i *       1.1055          *
*                                                                     *
*  =======================================================            *
*  Results of test tun #3                                             *
*                                                                     *
*  Quadratic Equation Root Solver Program                             *
*                                                                     *
*  Enter the quadratic equation's coefficients a, b and c             *
*  separated by a blank                                               *
*  2 8 6                                                              *
*                                                                     *
*  Two real roots at:      -1.0000                                    *
*                          -3.0000                                    *
*                                                                     *
*  =======================================================            *
*  Results of test tun #4                                             *
*                                                                     *
*  Quadratic Equation Root Solver Program                             *
*                                                                     *
*  Enter the quadratic equation's coefficients a, b and c             *
*  separated by a blank                                               *
*  4 4 1                                                              *
*                                                                     *
*  Multiple roots at      -0.5000                                     *
.))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

Design Exercises

1. Avoiding a Mostly Working Program.

Programs usually accept some kind of user input. Here we are dealing with numerical input data. When a specific program uses numerical data, the programmer must be alert for particular numerical values which, if entered, cause problems for the algorithm or method that is being used. The programmer must check for possible incorrect values and take appropriate actions. Sometimes the program specifications tell the programmer what to do if those “bad values” are entered; other times, it is left to the good sense of the programmer to decide what to do.

For each of the following, assume that the input instruction has been executed. You are to sketch in pseudocode the rest of the needed instructions.

a. The program accepts a day of the week from one to seven. It displays Sunday when the day is one; Monday, when two; Saturday, when seven.

Input a dayNumber

b. Housing Cost Program. When buying a house, the seller specifies the length and width of the outside of the home, the number of stories it has and the asking price. This program calculates

the actual cost per square foot of real living area within that home. Usually, 25% of the home area is non-liveable, being occupied by doors, closets, garages and so on. Using this program, a buyer can evaluate which home offers the best living area value. Using pseudocode, indicate what should be done to make this program work appropriately for all possible numerical inputs.

```
input the length, width, numberStories and cost
let grossArea = length * width * numberStories
let nonLivingArea = grossArea * .25
let liveableArea = grossArea – nonLivingArea
let realCostPerLiveableFoot = cost / liveableArea
output the realCostPerLiveableFoot
```

2. Comparison of Cereal Prices. Grocery store shoppers are often looking for the best value for their money. For example, a given type of cereal may come in several different sized boxes, each with a different price. A shopper wants to purchase the most cereal they can for the least money; that is, they want the best value for their money. A further complexity arises with coupons. Specific size boxes may have a coupon available to lower the total cost of that box. This program inputs the data for two different boxes and displays which one has the better value (most cereal for the least money). Write the rest of the pseudocode to determine for each box, the actual cost per ounce. Then, display the actual cost per ounce of each box and which is the better value, box1 or box2.

```
Input box1Weight, box1Cost, box1CouponAmount
Input box2Weight, box2Cost, box2CouponAmount
```

Stop! Do These Exercises Before Programming

1. Given the following variable definitions, what is the result of each of the following test conditions? Mark each result with either a t (for true or 1) or f (for false or 0).

```
     int x = 10, y = 5, z = 42;
____ a. if (x > 0)
____ b. if (x > y)
____ c. if (x == 0)
____ d. if (x == z)
____ e. if (x + y > z)
____ f. if (x / y == z)
____ g. if (x > z / y)
____ h. if (x > 0 && z < 10)
____ i. if (x > 0 && z >= 10)
____ j. if (x > 0 || z < 10)
____ k. if (x > 0 || z >= 10)
____ l. if (x)
____ m. if (!x)
```

2. Using the definitions in 1. above, what is the output of the following code?

```
if (z <= 42)
 cout << "Hello\n";
else
 cout << "Bye\n";
```

3. Using the definitions in 1. above, what is the output of the following code?

```
int t = y > x ? z : z + 5;
cout << t;
```

4. Correct all the errors in the following coding. The object is to display the fuel efficiency of a car based on the miles per gallon it gets, its **mpg**.

```
if (mpg > 25.0) {
  cout << Gas Guzzler\n";
else
  cout << "Fuel Efficient\n";
```

In the next three problems, repair the If-Then-Else statements. However, maintain the spirit of each type of If-Then-Else style. Do not just find one way to fix it and copy that same "fix" to all three problems. Rather fix each one maintaining that problem's coding style.

5. Correct all the errors in the following coding. The object is to display "equilateral triangle" if all three sides of a triangle are equal.

```
if (s1 == s2 == s3);
   {
   cout << "equilateral triangle\n";
   }
else;
   cout >> "not an equilateral triangle\n";
```

6. Correct all the errors in the following coding. The object is to display "equilateral triangle" if all three sides of a triangle are equal.

```
if (s1 == s2)
   if (s2 == s3)
      cout << "equilateral triangle\n";
cout >> "not an equilateral triangle\n";
```

7. Correct all the errors in the following coding. The object is to display "equilateral triangle" if all three sides of a triangle are equal.

```
if (s1 == s2) {
   if (s2 == s3) {
      cout << "equilateral triangle\n";
   }
   else {
      cout >> "not an equilateral triangle\n";
   }
}
```

8. Correct this grossly inefficient set of decisions so that no unnecessary decisions are made.

```
if (day == 1)
   cout << "Sunday\n";
if (day == 2)
   cout << "Monday\n";
if (day == 3)
   cout << "Tuesday\n";
if (day == 4)
   cout << "Wednesday\n";
if (day == 5)
   cout << "Thursday\n";
if (day == 6)
   cout << "Friday\n";
if (day == 7)
   cout << "Saturday\n";
```

9. Correct this non-optimum solution. Consider all of the numerical possibilities that the user could enter for variable **x**. Rewrite this coding so that the program does not crash as a result of the numerical value entered by the user. You may display appropriate error messages to the user. Ignore the possibility of the user entering in nonnumerical information by accident.

```
double x;
double root;
double reciprocal;
cin >> x;
root = sqrt (x);
reciprocal = 1 / x;
cout << x << " square root is " << root
     << " reciprocal is " << reciprocal << endl;
```

10. Correct this inherently unsound calculation.

```
double x;
double y;
cin >> x;
y = x * x + 42.42 * x + 84.0 / (x * x * x + 1.);
if (!y || y == x) {
 cout << "x's value results in an invalid state.\n"
```

```
 return 1;
}
```

Programming Problems

Problem Cs04-1 — Easter Sunday

Given the year inputted by the user, calculate the month and day of Easter Sunday. When the program executes, it should produce output similar to this.

Easter Sunday Calculator
Enter the year: 1985
Easter Sunday is April 7, 1985

The formula is a complex one and produces the correct day for any year from 1900 through 2099. I have broken it down into intermediate steps as follows. Frequently, Easter Sunday is in March, but occasionally it is in April. The following formula calculates the day of the month in March of Easter Sunday.

let a = year % 19
let b = year % 4
let c = year % 7
now start to put these pieces together
let d = (19 * a + 24) % 30
let e = (2 * b + 4 * c + 6 * d + 5) % 7
finally, the day of the month of Easter Sunday is
let day = 22 + d + e

However, if the day is greater than 31, then subtract 31 days and the resulting value in day is in April instead. But the equation is off by exactly 7 days if these years are used: 1954, 1981, 2049 and 2076. Thus, when the calculation is finished, if the year is one of these four, you must subtract 7 days from the day variable. The subtraction does not cause a change in the month.

Test your program on the following years — I have shown the day you should obtain in parentheses:

1985 (April 7)
1999 (April 4)
1964 (March 29)
2099 (April 12)
1900 (April 15)
1954 (April 18)
1981 (April 19)
2049 (April 18)
2076 (April 19)
1967 (March 26)

Problem Cs04-2 — Calculating Wages

Calculate a person's wages earned this week. Prompt and input the person's social security number (nine digits with no dashes), their hourly pay rate, the hours worked this week and the shift worked. The shift worked is 0 for days, 1 for second shift and 2 for the "graveyard shift". The company pays time and a half for all hours worked above 40.00. The additional shift bonus is a 5% for second shift and 15% for the graveyard shift. Format the output as follows:

```
Employee Number:      999999999
Hours Worked:             99.99
Base Pay:             $ 9999.99
Overtime Pay:         $ 9999.99
Shift Bonus:          $ 9999.99
Total Pay This Week:  $99999.99
```

Make the following test runs of the program.

```
Employee   rate  hours shift
123456789  5.00  40.00   0
123456788  5.00  40.00   1
123456787  5.00  40.00   2
123456786  5.00  60.00   0
123456785  5.00  60.00   1
123456784  5.00  60.00   2
123456783  5.00   0.00   2
```

Problem Cs04-3 — Scholastic GPA Results

The program inputs a student id number that can be nine digits long and their grade point average, GPA. The program is to display that student's status which is based only on their GPA. If the GPA is less than 1.0, the status is Suspended. If the GPA is less than 2.0 but greater than or equal to 1.0, then the status is Probation. If the GPA is greater than or equal to 2.0 and less than 3.0, the status is Satisfactory. If the GPA id greater than or equal to 3.0 and less than 4.0, then the status is Dean's List. If the GPA is 4.0, then the status is President's List. Display the results as follows.

```
    Id      GPA    Status
123456789   3.25   Dean's List
```

Make sure that no unneeded tests are made. That is, if you find that the GPA is that for Suspended, then do not additionally test for the other conditions. Once you have found a match, when finished displaying the results, do not subject that set of input data to additional test conditions.

Test your program with several test runs. The following should thoroughly test the program.

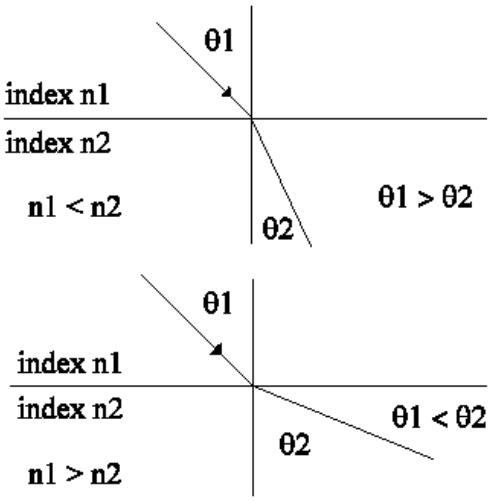
```
123456789 0.5
```

```
123456788 1.0
123456787 1.1
123456786 2.0
123456785 2.1
123456784 3.0
123456783 3.1
123456782 3.9
123456781 4.0
```

Problem Engr04-1 — Snell's Law — Optical Engineering

Snell's Law gives the angle that light is bent when it passes through a region with an index of refraction n_1 into another region with a different index of refraction n_2. An example is a light ray that passes through water in a crystal bowl. As the ray passes from the water through the clear crystal glass sides of the container, it is bent according to Snell's Law.

$n_1 \sin \text{angle}_1 = n_2 \sin \text{angle}_2$



When a ray passes from a region with a low index of refraction n_1 into a region with a higher index n_2, the exit angle is smaller than the entrance angle or the light bends toward the vertical. When passing from a region with a higher index of refraction into a region of lower index of refraction, the angle of exit is greater than the entrance angle or the angle bends away from the vertical. This is shown in the above drawing.

Write a program that calculates the exit angle of incidence angle_2, given the entrance angle of incidence angle_1 and the two indices of refractions, n_1 and n_2. Prompt the user to enter these three values; the angle input should be in degrees. Display the original input data along with the exit angle nicely formatted. The equation to be solved is

$$\theta_2 = \sin^{-1}\left(\frac{n_1}{n_2}\sin\theta_1\right)$$

Caution: if $n_1 > n_2$, then for some angles, the absolute value passed to the arcsine is greater than 1.0. This means that all light is reflected back in the direction it came from and none goes into the region two.

Test your program using a crystal bowl of water. The bowl is made of Crown Glass whose index of refraction is 1.52326. The water has an index of 1.33011.

Test 1 θ_1 is 30 degrees coming from the glass and going into the water
Test 2 θ_1 is 90 degrees coming from the glass and going into the water
Test 3 θ_1 is 90 degrees coming from the water and going into the glass
Test 4 θ_1 is 30 degrees coming from the water and going into the glass

Problem Engr04-2 — Power Levels

Decibels (dB) are often used to measure the ratio of two power levels. The equation for the power level in decibels is

$$dB = 10\log_{10}\frac{P_2}{P_1}$$

where P_2 is the power level being monitored and P_1 is some reference power. Prompt the user for the two power levels; then calculate and display the resulting decibels. You must guard against all ranges of numerical entries for the two power levels. Test your program with these inputs for P_1 and P_2.

```
1.0    5.0
1.0    50.0
496.64 1932.4
0.0    42.
```

Problem Engr04-3 — Formula Evaluation

Write a program to evaluate the following function for all possible numerical values of x that the user can input.

$$y(x) = \ln\frac{1}{1-x}$$

Show sufficient test runs to demonstrate that all possible situations are handled by your program.