PART IV

DECIDING

Often, you will have to check a condition and decide to act one way or another, depending on the result of that condition. In Part IV, you will learn how to incorporate decisions in your programs. You will also learn how to use a technique called *recursion*, in which you essentially work backwards toward the solution of a problem. You will also work on a short project to improve your skills in designing applications.

SKILL

TEN

INCORPORATING DECISIONS

- Deciding *if* you should do something
- Deciding *if* you should do this or *else* do that
- Specifying and testing conditions
- Breaking out of loops
- Designing loops—Part 2
- Switching among alternatives

In SKILL ten, you will develop your ability to incorporate decisions in your programs. Decisions choose to execute one piece of program or another depending on a condition. You will learn how to specify conditions and how to use conditions in an `if` statement, so that you can decide the appropriate sequence to execute.

You will also learn how to interrupt a loop if a certain condition arises, and you will improve your skills in designing loops. The `Robot` objects will assist you with these skills.

Using the *if* Statement: to Do or Not to Do

Most decisions involve a condition, and then an action. For example, If the weather is rainy, take the umbrella. In this case, the condition is *the weather is rainy* and the action is *take the umbrella*. The action is executed only if the condition is satisfied. The condition is the actual basis for the decision. In the example above, all we care to know is whether the weather is rainy.

When you write programs, you use a similar decision process. Most decisions will be based on a condition, and, depending on this condition, you may want to perform some action. The `if` statement is the most important decision mechanism in programs.

Let's start with the simplest kind of decision. You may simply want to decide whether to do something. You may be already familiar with the following example:

If you have money:
    Go to the movies.

which means that the action *go to the movies* will only be executed if the condition is true *(you have money)*. Otherwise, nothing will happen, and you will simply execute the next statement, which would probably be *go to sleep*. This process is shown in Figure 10.1.

A more complete description of the problem could be as follows:

If you have money:
    Go to the movies.
Go to sleep.

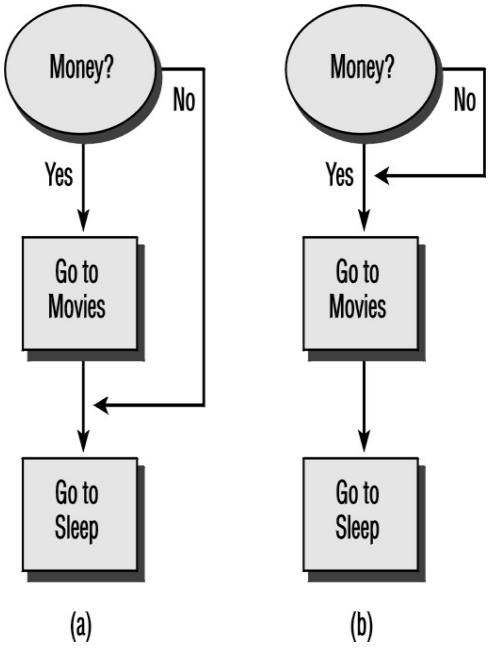


Fig 10.1 Going to the movies 1

It is important to understand that, in this case, you will go to sleep either way. The difference is that the action *go to the movies* either will be executed or will not be executed. How do we know this? Because of the indentation! If we had the following directions:

If you have money:
Go the movies.
Go to sleep.

without indentation, you might think that you should check your pocket to see if you have any money, but no matter what the result is, you would go to the movies and then go to sleep. This is because, in our convention, the compound statement affected by the condition should be indented. In a C++ program, you must also include braces.

The kind of decision we are considering is that either you do something or you don't. This is similar to what you've had Tracer do already. You wanted her to step ahead or not, depending on whether she saw a wall. That procedure was enclosed in the following repetition:

```
while (Tracer.seenowall())
{
     Tracer.step();
}
```

`while` decides whether to repeat the loop, depending on the presence of a wall ahead. Did you notice that? What if you want her to take just one step, instead of walking until she finds a wall? Basically, you would explain something like the following:

If you see no wall:
    Step ahead.

Tracer should check the condition *see no wall,* and, if this is true, she should take the action *step ahead*. What if the condition is not true (if she, indeed, sees a wall)? Well, nothing would happen! She would not step ahead, and the program would just go on to the next statement.

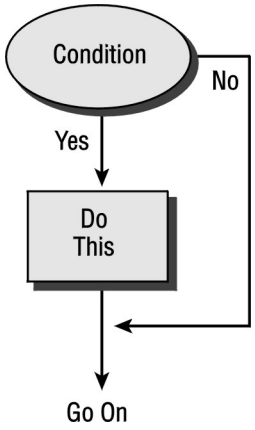


Fig 10.2 Deciding if

In C++, you use the `if` statement. The action of this statement is shown in Figure 10.2.

The `if` statement looks like the following:

```
if ( condition )
 {
     statements  // Do this
 }
```

Or, in our example:

```
 if(Tracer.seenowall())
{
     Tracer.step();
}
```

Pay attention to the format in which you write the statement:

- Use the keyword `if`.
- Write a condition, which is a relational expression inside parentheses.
- Indent and enclose in braces the sequence of statements that you want executed.

There is no semicolon after the condition, nor is there one after the compound statement. This is a common beginner's mistake—if you include a semicolon after the condition, the compiler thinks that you want to do *nothing*. A semicolon after the compound statement is harmless.

If the sequence to be executed consists of only one statement, you don't need to enclose it in braces. However, I strongly encourage you to develop the habit of using braces.

For example, if you want Tracer to see whether there is a wall ahead and mark that square (and do nothing otherwise), you could write a piece of program such as the following:

```
 if (Tracer.seewall())
{
     Tracer.mark();
}
```

As you can see, if there is no wall ahead, no marking will take place.

Example—In the Maze 1

Let's develop a couple of functions that may be used with our robot. One of these functions, `go`, will make the robot step ahead after it checks whether the square is empty. If the square is not empty, no action will be taken. The second function, `crossing`, will examine the squares adjacent to the robot, and will compute how many of them are free. In other words, `crossing` will compute how many directions (N, S, E, W) the robot can follow.

The algorithm for `go` is very simple:

If there is no wall ahead:

Step ahead.

Notice that no action is to be taken if there is a wall ahead. Here is one implementation:

```
void go()
 {
    if(Tracer.seenowall())
    {
       // Do this only if the next square is clear:
       Tracer.step();
    }
 }
```

To make this function more interesting, we may include the following:

- Mark the square with a different color.
- Have Tracer say "OK" to indicate the successful move.

If you use the light-blue color (code=4), the modified function is as follows:

```
void go()
{
   if(Tracer.seenowall())
   {
      // Do this only if the next square is clear:
      Tracer.mark(4);
      Tracer.step();
      Tracer.say("OK");
   }
}
```

The other function, `crossing`, is also simple. The algorithm consists of using a variable to count the available directions (`howmany`) and looking at the next square to determine whether it is occupied. If it is not occupied, we add one to `howmany`. If this procedure is applied to the four available directions, `howmany` will contain the number of free directions.

USING INTEGERS AND PARENTHESES

If you declare the `Robot` object followed by an integer enclosed in parentheses (`Robot Tracer (1);`), it will place the Robot in a maze. The position and direction, as well as the location of the walls, are unknown to you at that time. You may then have fun exploring the maze.

If the `Robot` declaration does not include an integer enclosed in parentheses (`Robot Tracer;`), the Robot is placed in an empty room, facing east.

Declare only one `Robot` in your programs; also, declare the `Robot` globally instead of in a function. This will avoid the need to pass the `Robot` as an argument.

Try This for Fun…

- Write the function `int crossing()` to determine how many directions are available to the robot. Test this function with the following program:

```
#include "franca.h"
 Robot Tracer(1);
 void mainprog()
 {
    Tracer.say(crossing());
 }
```

Example—At the Store 1

Let's now consider the problem of computing the change for a merchandise sale. The basic problem is illustrated below.

```
#include "franca.h"
void mainprog()                        // c4change.cpp
{
  float price,amount,change;
  float tax=0.08;
  amount=ask("Enter the amount:");
  price=ask("Enter the price")*(1+tax);
  change=amount-price;
  Cout<<change;
}
```

This program reads an amount and a price, and computes the change after including the sales tax. The program outputs the change to be given to the customer. Notice that the program does not check whether the amount tendered is sufficient to cover the price!

A possible improvement to this program would be to check whether the amount is sufficient, and to warn the clerk when it is not. For example:

```
#include "franca.h"
 void mainprog()
 {
   float price,amount,change;
   float tax=0.08;
   amount=ask("Enter the amount:");
   price=ask("Enter the price")*(1+tax);
   if(amount<price)
   {
      Clock time;
      Cout<<"This is not enough!";
      time.wait(5);
    }
   change=amount-price;
   Cout<<change;
 }
```

The only change is that a message is displayed if the amount is insufficient. A clock was needed to hold the message for 5 seconds, so that it could be read.

Using the *if/else* Statement: to Do This or to Do That

The other kind of decision you may want to use is to choose between two alternatives—to do either this or that.

Take your ATM card.
If the store is open:
    Buy a dozen eggs.
    Get twenty dollars.
Else:
    Get twenty dollars from the ATM.
Fill the gas tank.

Depending on whether the store is open, you will take one of two alternatives:

- Buy a dozen eggs and get twenty dollars (at the store).

or

- Go to a 24-hour ATM to get twenty dollars.

You will not do both things!

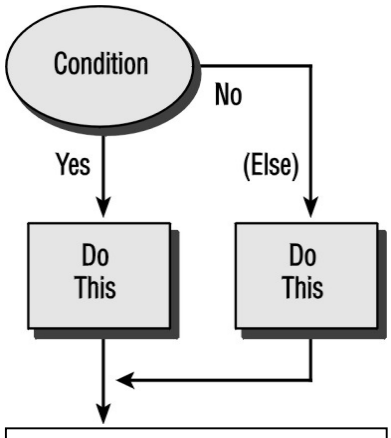


Fig 10.3 Deciding if or else

No matter which action is taken, you are supposed to fill the gas tank, because this action is not affected by the condition. Essentially, in this case you can specify two alternative actions: either do this or do that, as shown in Figure 10.3.

The syntax is similar to that of the plain `if` statement that we just saw. All you have to do is to append the keyword `else`, followed by the action you want to perform if the condition fails.

```
 if ( condition )
{
      statements // Do this
}
else
{
      statements // Do that
}
```

For example, suppose you want Tracer to mark a square in black or green, depending on whether she sees a wall:

```
int black=7,green=2;
if (Tracer.seewall())        // Check for a wall ahead. If so:
{
     Tracer.mark(black);     // Mark in black
}
else
{
     Tracer.mark(green);     // Else: Mark in green
}
```

Of course, since each alternative action consists of only one statement in this case, you don't really need the braces. You could write this program as follows:

```
if (Tracer.seewall()) Tracer.mark(black);
else Tracer.mark(green);
```

Try This for Fun…

- Use Tracer to walk down a corridor and locate the first corridor to the left. Remember that she can only see whether there is a corridor to the left if she turns left to see it! When she finds the corridor, tell her to stay there and report how many steps away the corridor is located.

Example—In the Maze 2

We'll now revise the program `c4search.cpp` to incorporate a better visual interface. All we need to do is to mark the squares in which a wall was found in a different color. A simple modification of the `go()` function will do the trick. If you examine the structure of the `if/else` construct:

```
if ( condition )
 {
    // Do this
 }
 else
 {
    // Do that
 }
```

you will notice that the *do this* part already exists. We need only to include the *do that* part. Can do you this by yourself? Sure you can! (See the exercise below.)

Try This for Fun…

- Modify the `go()` function in `c4search.cpp` to perform the following actions if the next square contains a wall:

  Mark the next square in black.
  Say "OUCH!"

Example—In the Maze 3

Another interesting example is to have Tracer walk down a corridor and mark the squares that are empty and the ones that are occupied by a wall. To do this, you may tell Tracer to repeat the following sequence in every square:

Turn left and see if there is a wall.
If so:
    Mark the square in black.
Else:
    Mark it in green.
Face the original direction.
We can describe this problem as follows:

Repeat until you see a wall:
    Check your right.
    Check your left.
Step ahead.

Of course, we must explain what we mean by *check your right* and *check your left*! Here is how you check your right:

Turn right.
If you see wall:
    Mark the square in black.
Else:
    Mark the square in green.
Turn left.

You can figure out how to check the left.

Can you see, once again, the role of `if/else`? Depending on the existence of a wall, Tracer either marks the square in black or marks the square in green. She has to choose one thing or the other. If the condition is true, the action following `if` is executed, and the action following `else` is skipped. If the condition is not true, the action following `if` is skipped, and the action following `else` is executed.

The piece of program to check your right could look like the following:

```
Tracer.right();
if (Tracer.seenowall())
{
     Tracer.mark(green);
}
else
{
     Tracer.mark(black);
}
Tracer.left();
```

You might want to consider creating a function to `check`:

```
void check()
{
     if (Tracer.seenowall()) Tracer.mark(green);
     else Tracer.mark(black);
}
```

Your program could work as follows:

```
#include "franca.h"
 Robot Tracer (1);
 void mainprog()
 {
      while(Tracer.seenowall)      // Repeat until wall is seen
      {
           Tracer.left();          // Check your left
           check();
           Tracer.right();
           Tracer.right();         // Check your right
           check();
           Tracer.left();
           Tracer.step();          // Step ahead
      }
 }
```

Example—At the Store 2

At this point, we may also reexamine the program that computes change, `c4change.cpp`. This program was presented earlier in the skill, and was modified to check whether the amount tendered was sufficient to purchase the items.

A better solution to this problem includes a *do this or do that* alternative. In other words, if the amount tendered is insufficient, the sale should not be completed. One possible solution is illustrated below.

```
#include "franca.h"
void mainprog()
{
  float price,amount,change;
  float tax=0.08;
  amount=ask("Enter the amount:");
  price=ask("Enter the price")*(1+tax);
  if(amount<price)
   {
     Cout<<"This is not enough!";
   }
  else
   {
     change=amount-price;
     Cout<<change;
   }
}
```

In this case, either the message *This is not enough!* will be displayed, or the change will be displayed—but not both!

Specifying Conditions

Conditions are used in decision and repetition statements. We have already experimented with simple conditions that compare two numbers. For example:

```
count<=15
```

checks whether the count is less than or equal to 15. Conditions are based on the relationship between values, and are expressed by means of relational expressions.

Relational Expressions

The simplest relational expression consists of two items with a relational operator between them. The items can be the following:

- Variables—for example, the following checks whether `count` is equal to `howmany`:

  ```
  count==howmany
  ```

- Constants (relational expressions with two constants do not make sense, but they can be used)—for example, the following checks whether `howmany` is *not* equal to zero:

  ```
  howmany!=0
  ```

- Results returned by a function—for example, the following checks whether the time in `myclock` is less than or equal to 20:

  ```
  myclock.time()<=20
  ```

Objects can also be used in a relational expression, but only if the relational operator is defined for that class of objects.

Unlike the assignment operator, there is no difference whether you use an item to the left or the right of the relational operator. For example, the same results will be achieved if we express the condition `count<=15` as follows:

```
15>count
```

Relational Operators

The relational operators, which were seen in SKILL five, are the following:

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

Results

The computer evaluates the relational expression and generates a result. For practical purposes, the relation is either true or false. For example, if the current value of `count` is 13, the following expression:

```
count<=15
```

is true.

Once again, remember the most dangerous expression—`if (a=b)`. It is not a condition—it is just an assignment. A condition uses two equal signs (`==`)!

Compound Expressions

In many cases, we need to base our decision on more than one expression. For example, an athlete may be dismissed if she has performed 50 jumping jacks, or if she has been exercising for more than 5 minutes. In this case, there are two conditions:

```
count>=50
time>=300
```

When two or more conditions are used, we must explain how they are supposed to be combined. In this case, we want one condition or the other to be satisfied. In other cases, we might want both conditions to be satisfied (one *and* the other).

Logical Operators

Logical operators are used to explain how to combine two or more conditions. The logical operators are the following:

- &&—the `and` operator (one condition and the other must be true)
- ||—the `or` operator (one condition or the other, or both, must be true)
- !—the `not` operator (inverts the result of a relational expression)

The `not` operator is unary; it may precede an expression, and it will invert its value. The other operators are binary; they are placed between two expressions. The expressions must always be enclosed in parentheses.

Here are some examples:

```
(count>=15)||(myclock.time()>300)
```

Either `count` is greater than or equal to 15, or `time` is greater than 300.

```
(count>=15)&&(myclock.time()>120)
```

`count` is greater than or equal to 15 and `time` is greater than 120.

```
(count>15)&&(!(myclock.time()<240))
```

`count` is greater than 15 and `time` is *not* less than 240.

Priority

When more than two expressions are evaluated, it is important to know the operator priority:

- The `not` operator has the highest priority.
- The `and` operator has the next priority.
- The `or` operator has the last priority.

Parentheses can be used to specify or to reinforce priority. For example:

```
(myclock.time()>300)||(myclock.time()>120)&&(count>=50)
```

In this example, the `and` operation is done first, followed by the `or`. If this expression is used to dismiss an athlete, the athlete will be dismissed in two cases:

`time` is greater than 300

or

`time` is greater than 120 and `count` is greater than or equal to 50.

Nested Decisions

Some situations require you to check more than one condition, so that some action can be performed. This happens when one of the statements to be executed in an `if` is another `if`. Consider the following situation:

```
Tracer.right();
    // Check the right side:
    if(Tracer.seewall())
    {
       Tracer.left();
       Tracer.left();
       // Check the left side:
       if(Tracer.seewall())
       {
          Tracer.say("corridor");
       }
    }
```

In the situation above, we have two levels of nesting. If the first `if` fails (no wall), the second `if` is not executed. This piece of program determines whether the squares on both sides of Tracer have walls. Notice that Tracer first turns to the right to check the right side. If there is no wall, there is no need to check the left side, and the test is done. If a wall appears in this right square, Tracer must turn to the left square (two turns will be needed at this point) and check it. If a wall is detected, Tracer can "say" that she is in a corridor in which both left and right squares have walls.

It is a good idea to imagine yourself as the robot and to try to understand exactly how this algorithm works.

The more nested conditions you use in your program, the more complex your program will become. Correct and clean indentation, as well as sufficient comments, will significantly help to keep your programs readable. If complexity persists, you should consider resorting to functions.

A major inconvenience of the C++ programming language is that it makes no provision for the compiler to determine whether an `if` statement has an `else` clause.

For example, consider the following:

```
k=0;
 if (a<0)
      if (b<0) k=1;
 else k=2;
```

and

```
k=0;
      if (a<0)
           if (b<0) k=1;
           else k=2;
```

Since the computer does not understand indentation, both pieces of program are identical in the eyes of the compiler. However, the indentation leads us to believe that the results would be different. In the first case, if `a` is greater than or equal to zero, `else` is executed and `k` is assigned the value 2. In the second case, the `else` clause is associated with the inner `if`, and, therefore, if `a` is greater than or equal to zero, the inner `if` is skipped, and the value of `k` remains zero.

Of course, only one of the alternatives is correct, because, as we mentioned, they appear exactly the same to the compiler. What does the compiler do in this case?

The `else` clause always matches the closest `if`.

The second alternative is the correct one, because the `else` clause is closest to the second `if`.

If you want to make `else` apply to the first `if`, you have to include an empty `else`, as illustrated below.

```
k=0;
 if (a<0)
      if(b<0) k=1;
      else ;
 else k=2;
```

Another, more elegant, approach is to clearly delimit the `if` without the `else`.

```
k=0;
 if (a<0)
      {
       if(b<0) k=1;
      }
 else k=2;
```

Breaking Out of Loops

Sometimes you may be repeating a loop, and, for some reason, you decide to interrupt the repetition.

Suppose that our good friend Sal is told to perform 50 jumping jacks in his fitness class. Well, he is happily jumping—1, 2, 3, 4—when all of a sudden the fire alarm sounds! What do you think he should do? Finish the jumping jacks, and then run away? Of course not! Sometimes we want to interrupt a repetition and break out.

The *break* Statement

The `break` statement serves exactly this purpose. This statement allows you to escape the current loop as if the repetition was finished at that point. Figure 10.4 shows this situation.

Suppose that you set a loop to repeat a certain number of times. However, in the loop, you test for another condition—*Is it too late? Is the building on fire?*—and then you decide to break the loop. You will not only stop repeating, you will also skip any remaining part of the loop that has not been executed.

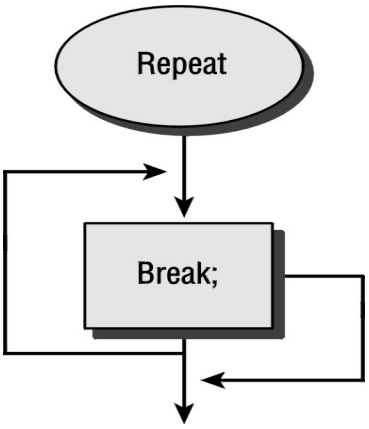


Fig 10.4 Breaking out

To illustrate this, why don't we have Sal do some exercises, and then check the time he has spent. This can be done as follows:

Repeat 50 times:
    If time is more than 50 seconds:
        Stop.
    Do a jumping jack.

The following could be the program:

```
athlete Sal;
Clock watch;
for (int times=1;times<=50;times++)
{
    if (watch.time()>50.) break;
    JumpJack(Sal);
}
```

Did you miss an `else`? You might have expected the decision to use an `if/else` structure, instead of the plain `if`. After all, `JumpJack` will be executed only if the time is less than 50 seconds. In other words, you might have thought of writing the program as follows:

```
athlete Sal;
Clock watch;
for (int times=1;times<=50;times++)
{
    if (watch.time()>50.) break;
    else JumpJack(Sal);
}
```

This solution is correct. When the `break` occurs, whatever statements are left until the end of the loop (marked by the closing brace) are skipped. The `JumpJack` will be performed only if the `break` is not performed. There is no need for the `else` in this case.

An interesting example tells Tracer to step 10 times:

Repeat 10 times:
    Step.

However, since we don't want her to bump into a wall, we want her to stop this repetition if she happens to see a wall ahead:

Repeat 10 times:
    If you see a wall:
        Stop repeating.
        Step.
Done!

What do we want her to do? We want her to ignore the condition of the repetition, and to go straight to the next statement after the repetition—*Done!* Use the `break` statement again. The program would look like the following:

```
int stepcount;
for (stepcount=1;stepcount<=10;stepcount++)
{
     if(Tracer.seewall())
     {
          break;
     }
     Tracer.step();
}
```

The *continue* Statement

In some repetitions, you may decide to suspend the current pass of the loop, but to continue to repeat the next pass. This is the role of the `continue` statement. `continue` is similar to `break`—they both interrupt the repetition. Unlike `break`, `continue` skips only the remaining part of the current loop, but it continues with the next iteration.

If you are familiar with board games, you may compare `break` with being kicked out of the game altogether. `continue` sends you back to the start—go directly to jail, do not pass *Go,* do not collect $200—but you resume play on your next turn.

As an example, suppose that one of our athletes exercises by repeating the following sequence 50 times:

jumping jack
leftright

If we assume that we have defined the functions as follows:

```
JumpJack(athlete)
leftright(athlete)
```

the program could look like the following:

```
athlete Kim;
Clock watch;
for (int times=1;times<=50;times++)
{
   JumpJack(Kim);
   leftright(Kim);
}
Kim.say("Done!");
```

Now, suppose that Kim wants to check the time, and, if the time exceeds 15 seconds, to skip the second part of the exercise (the `leftright`). The new listing below includes an `if` that checks the time and skips the remaining part of the loop:

```
athlete Kim;
Clock watch;
for (int times=1;times<=50;times++)
{
   JumpJack(Kim);
   if (watch.time()>15.) continue;
   leftright(Kim);
}
Kim.say("Done!");
```

Each time Kim goes through the loop, she performs a jumping jack and checks the time. If the time is still less than 15, she proceeds to the end of the loop, performing the `leftright`. However, if the time is greater than 15, the `continue` statement is executed. The remaining steps in the loop are skipped, but the variable `times` will be incremented, and the next pass of the loop will take place. Since the time will always be greater than 15 from that point on, Kim will only perform `JumpJacks` and will skip the `leftright`. If a `break` is used instead of a `continue`, the complete repetition will be interrupted and no more `JumpJacks` or `leftrights` will be performed.

Designing Your Own Loops—Part 2

We missed using decisions and the `break` and `continue` statements in "Designing Your Own Loops—Part 1" in SKILL eight. By using these, you can either skip part of the loop, or exit it completely. The loop can be interrupted at any point when you use the `break` statement. The loop does not have to be completed every time.

As an example, let's refer to the program `c3avgrd.cpp`, in which we computed the average grade in a class. The main loop was implemented as follows:

```
for (int which=0;yesno("input another grade?"); which++)
{
  grade=ask("Enter next grade:");
  total=total+grade;
}
```

Suppose that your teacher did not like the idea of having to reply to the question *input another grade?* all the time. After all, if there were 40 students in class, this would represent an extra 40 mouse clicks!

An alternative is to use a sentinel value. For example, since grades are supposed to be all positive, you could reserve a negative value to signal that there are no more grades to be input.

The new loop could then become the following:

```
for (int which=1;; which++)
{
  grade=ask("Enter next grade:");
  if(grade<0) break;
  total=total+grade;
}
```

This causes immediate interruption of the loop when a negative grade is input. This negative grade is read, but it will not be added to the total (`break` occurs before addition). The variable `which` will not be incremented either. Notice that I changed the initial value to 1, instead of zero. I did this because once `break` is executed, the counter will no longer be incremented. If you set the initial value to 1, it will result in correct values for `total` and `which`.

Notice that no condition is necessary in the `for` statement. However, make sure that at some point, some statement causes the loop to end!

You may feel tempted to use the sentinel value in the `for` statement. For example:

```
for (which=1;grade>=0; which++)
{
  grade=ask("Enter next grade:");
  total=total+grade;
}
```

may seem right, because the loop will be executed as long as the grade is not negative. This is not true. Remember that the condition is tested before the loop is executed. The negative grade will be read and added to the total, and the control variable (`which`) will be incremented. Only then will the condition be checked for the next iteration and the loop be stopped! Furthermore, when you execute the loop for the first time, you don't know if `grade` has any valid contents.

Switching among Several Alternatives

In certain situations, instead of having to choose between two alternatives, we have to choose one among several. For example, if we want an athlete to assume a different position according to the value contained in a variable `code`, we could do the following:

```
if(code==1) Kim.ready();
if(code==2) Kim.up();
if(code==3) Kim.left();
if(code==4) Kim.right();
```

`switch` uses one variable to decide which action to take (see Figure 10.5).

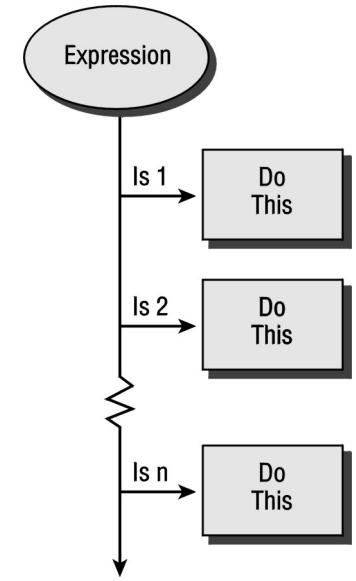


Fig 10.5 Switching

It works as follows:

Use the value to choose one case from the list below:

Case—the value is 1:
ready!
End of this case.
Case—the value is 2:
up!
End of this case.
Case—the value is 3:
left!
End of this case.
Case—the value is 4:
right!
End of this case.

End of the list of cases.

`switch` includes a header and a list of cases. (Figure 10.5 illustrated the procedure.) The general form of `switch` is as follows:

```
switch ( expression )
 {
     list of cases
 }
```

in which *expression* is any expression that returns an integer value (`int`, `long`, or `char`) and the *list of cases* (enclosed in braces) consists of any number of cases that have the following form:

```
case integer value:
     statements
break;
```

The effect of the `switch` statement is as follows:

- The expression is evaluated, and the resulting value (which must be an integer) is checked for a match with the integer values listed in the `case` labels.

- If a match is found, the sequence of statements that follows the label is executed until a `break` is found.

Example—Thank You for Calling!

To illustrate the use of `switch`, let's consider the problem of an international communications carrier that wants to write a message—*Thank you*—in the language that is spoken in the country they are calling.

You can write a program that requests the country code (for international direct dialing). Since there are many alternatives, it is impractical to use `if`. On the other hand, `switch` is very convenient. Once you have the country code, use it to `switch`:

```
switch (country_code):
 {
    case   1:
        message.say("Thank you");
    break;
    case   34:
        message.say("Gracias");
    break;
 ...
```

To make the problem even more interesting, consider the case of Switzerland. Each of its regions typically uses a different language. No problem! Just use the area code, as well. The complete program is shown below.

```
#include "franca.h"           // c4switch.cpp
 void mainprog()
 {
   Box display;
   int country,area;
   while(yesno("Shall we continue?"))
   {
        country=ask("Enter the country code:");
        area=ask("Enter the area code:");
```

```
            switch (country)
            {
                 case 55:          // Brazil
                 case 351:         // Portugal
                      display.say("Obrigado");
                 break;
                 case 81:          // Japan
                      display.say("Arigato");
                 break;
                 case 82:          // Korea
                      display.say("Gamzahamnidah!");
                 break;

                 case 52:          // Mexico
                 case 34:          // Spain
                 case 54:          // Argentina
                           display.say("Gracias!");
                 break;
                 case 41:          // Switzerland
                     switch (area)
                     {
                         case 21: // Lausanne
                         case 41: // Lucerne
                         case 22: // Geneva
                           display.say("Merci");
                         break;
                         default:
                           display.say("Danke");
                     }
                 break;

                 case 49:          // Germany
                 case 43:          // Austria
                      display.say("Danke!");
                 break;
                 default:
                      display.say("Thank you!");
            }
       }
  }
```

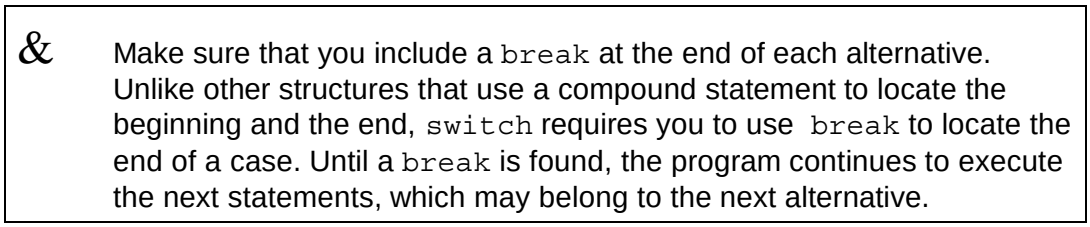
It is important that you note the following:

- The statement begins with the keyword `switch`, followed by an integer variable (or any expression that results in an integer value) in parentheses. This is the value that determines which action will be executed.
- The set of cases is enclosed in braces ({}).
- Each case is labeled with one possible value (expressed by a constant) and ends with a `break` statement. If you omit (or forget) a `break` statement between two cases, the *breakless* case will proceed until a `break` is found. In our particular problem, this comes in handy. We may group all countries that use the same language and end them with the same `break`.
- The label consists of the keyword `case`, a constant that expresses the selected value, and a colon. The label can be written in the same line as the first statement in the case. For example, the following is also correct:

  ```
  case 51: message.say("obrigado");
  ```

- If the value that is used for switching does not match any of the listed cases, no action will be taken.
- It is possible to specify a default case, which will be executed if the value does not match any of those listed. The default case is labeled with the keyword `default`, as in the following example:

```
default:
     message.say("Thank you");
```



Are You Experienced?

Now you can…

Use `if` to decide on the right piece of program to execute

Use `if/else` to decide between two alternatives

Build a condition on which you can base your decisions

Break out of loops

Design more complex loops

Use a `switch` statement to choose one among several alternatives

SKILL

ELEVEN

USING RECURSIVE FUNCTIONS

- Understanding recursive algorithms
- Creating recursive functions

Recursion is a technique that solves a problem by working as if a simpler, similar problem has already been solved (although it has not). The question then becomes, Who is going to solve the simpler problem? This is where recursion really takes over—it calls the same function (itself), but now it requests the solution to the simpler problem. What happens if the simpler problem is still not easy to solve? No problem, it just keeps calling itself!

As you use the same technique to solve the simpler problem, you will postpone the solution of an even simpler problem again and again. The simple problem becomes simpler each time—until it becomes trivial. What do you do then? You go back and solve each of the other problems whose solution was postponed.

This skill will be developed with the aid of the `Robot` object.

Understanding Recursive Algorithms

Recursive algorithms use themselves to reach an answer. Consider the problem in which you want to measure how many steps the robot has to take to reach a wall. A recursive answer to this problem would be as follows:

This is how you compute *distance:*

If you see the wall, the answer is zero.
Otherwise:
Step ahead.
The answer is 1+*distance.*

"Master, how many steps to the top of the mountain?" "Take one step now—you will be one step closer."

When the expression *1+distance* is reached, the result cannot proceed until *distance* is known. The computation is then suspended, so you can compute the new distance. When this value is found, you add one to it to find your answer. Let's try to understand how this algorithm works.

When you are told to compute the distance, the first thing you do is to check whether you are already facing a wall, in which case the distance is zero. However, if you are not facing a wall, you have to step ahead—your answer will be one plus the new distance. You could say that this is pretty obvious. But how do you compute the new distance? Just interrupt your computation and look in the algorithm again to check whether you are facing a wall—but remember to proceed when you obtain the result!

At some point, the wall will be reached, and the result *zero* will be returned. You then resume the computation that was interrupted and add one to the distance from the next square (which was found to be zero). The new result (one) will be returned, and the previous computation,

which was interrupted, can now resume. Several algorithms can be best expressed by means of recursion.

The distance from a given square equals the distance from the next square plus one.

Creating Recursive Functions

C++ supports recursive functions—functions that can call themselves. Here is one example:

```
int distance()
 {
   if(Tracer.seewall())return 0;
   Tracer.step();
   return 1+distance();
 }
```

This example is an implementation of the recursive algorithm to compute the distance to a wall, as discussed above. The last statement returns the value of the expression `1+distance()`. However, `distance()` is a function call to the same function (the function calls itself!). The only difference is that now the robot is one step closer than it was before.

At this point, you are strongly encouraged to visualize yourself as the robot and follow this algorithm to compute the distance, in steps, to the nearest wall. It would also be useful to have a function that shows the actual step count while the computation proceeds. The following listing is a possibility:

```
int distance()
{
  if(Tracer.seewall())return 0;
  Tracer.step();
  Tracer.say(1+distance());
  return 1+distance();
}
```

The function above seems to do the job, but it is inconvenient. Notice that we are invoking `distance()` twice. Suppose that you are the robot. Would you walk to the wall to compute the distance, say it, then walk back to return the result? If the `distance()` function leaves the robot next to the wall, this function will not work correctly, because the second time you call `distance()`, the result will be one. Try it! You may check this with the program `c4dist1.cpp`.

It is a good idea to store the result in a variable, so we do not have to invoke the function twice.

```
int findwall()
{
  int steps;
  if(Tracer.seewall())return 0;
  Tracer.step();
  steps= 1+findwall();
  Tracer.say(steps);
  return steps;
}
```

If you execute this function, you will notice the following things:

- The values of the steps are "said," *1, 2, 3….*
- The values of the steps are "said" only after the robot reaches the wall. Why?

How Does a Recursive Function Work?

Every time a function is called, the computer keeps track of where it is in the program's execution, and then it starts to execute the function. Any variables that were declared in the function are then created, and the arguments are copied to the parameters (unless they are passed by reference). When a function calls itself, it is as if the function is frozen at that time, and a new copy of the same function starts to execute.

The `findwall` function illustrates this process. Let's suppose that Tracer is located three squares away from the wall when the function is first called. Since there will be further calls, let's denote this as the first instance.

First instance: When the function is first invoked (by another program), the variable `steps` is created, and Tracer is in square number 1. Since she is still away from the wall, the function instructs her to step ahead. Tracer is now in square number 2.

Next, the function computes the value of `steps` as `1+findwall`. The recursion starts here. The remaining statements in the function are not executed. The function will be frozen with all the current variables until this result can be computed. The variable `steps` still does not have a value assigned to it. The function is called again (the second instance starts).

Second instance: A new instance of the function is started.
Another variable `steps` is created, and Tracer is in square number 2. Since she is still away from the wall, the function instructs her to step ahead. Tracer is now in square number 3.

Next, the function computes the value of `steps` as `1+findwall`. The recursion takes place again. The remaining statements in the function are not executed. (Wow! Do you remember doing the same thing in the previous issue of the function?) This issue of the function will now be frozen until this result is available. The second version of the variable `steps` also does not have a value assigned to it. The function is called once more (the third instance starts).

Third instance: A new instance of the function is started. Another variable `steps` (this is the third version) is created, and Tracer is in square number 3.
Since she is now next to the wall, the function terminates, and then returns the value *0* as a result. This terminates this instance of the function and yields a result, so the previous version can continue. Nothing is assigned to the variable `steps` (the computer now returns to the second instance).

Second instance: The computer continues the computation.
The second version of the variable `steps` is now assigned a value of `1+0`. Tracer sends this value to the screen, and the function returns this value (which happens to be 1). This instance of the function terminates and yields a result, so the previous version can continue (the computer finally returns to the first instance).

First instance: The computer continues the computation. The original version of the variable `steps` is now assigned a value of `1+1`. Tracer sends this value to the screen, and the function returns this value (which happens to be 2). The first instance of the function terminates, and this result is passed to the original calling program.

Improving the Algorithm

Something still makes me unhappy about this function. Every time we tell the robot to take a measurement, she walks to the wall and stays there! Couldn't she come back to the original place?

The crucial elements of this function are when we compute the value of `steps` and then invoke the function `findwall`, because the computer must do the following things:

- Suspend the work and remember where to resume.
- Restart the function to compute the distance.
- Resume where work was suspended when the result is available.

Well, one thing is certain—if we want her to come back to the place where she started, we better tell her to step back again after the measurement. It is as if we undo the steps after we find the result.

At this point, we could have a function that looks as follows:

```
int distance()
{
  int steps;
  if(Tracer.seewall())return 0;
  Tracer.step();         // Move forward
  steps= 1+distance();   // Compute distance
  Tracer.say(steps);     // Display current distance
  Tracer.step();         // Step back!
  return steps;
}
```

Did we forget to tell her to turn back? After all, she is moving toward the wall. Yes, we have to tell her to turn back. When shall we tell her?

You may think it is a good idea to tell her to turn back before she steps back—it is not! Notice that she only has to turn back once. If she turns back before she has taken the step back, she will end up turning several times. (Try it! It is fun.)

The appropriate thing to do is to have her turn back when she finds the wall. From then on, every time she has a measurement, she is already facing the return direction.

A final version of this function could then be as follows:

```
int distance()
{
  int steps;
  if(Tracer.seewall())
  {
      Tracer.right();
      Tracer.right();
      return 0;
  }
  Tracer.step();        // Move forward
  steps= 1+distance(); // Compute distance
  Tracer.say(steps);    // Display current distance
  Tracer.step();        // Step back!
  return steps;
}
```

A version of this function that is slightly more elaborate can be found in the program `c4dist2.cpp`.

Recursive Function Calls Use Memory

A recursive function "remembers" all the values as they were before the recursive call. All variables that were declared in the function are re-created in a new version, leaving the previous version intact. The same is true of any parameters passed by value, which are like new variables that are re-created every time the function is called. In addition, the function remembers exactly what it was doing, so it can later resume.

Trade-Offs in Recursion

Recursive algorithms are good because they may express a simpler way to solve a problem. However, there are some inconveniences of which you should be aware.

Every time a function is called:

- Memory space is allocated to accommodate any variables or objects that were declared in that function.
- Arguments and returned values also need memory space. In the `findwall()` example, a variable `steps` is declared in the function.
- The computer has to keep track of the place in the program where operation should resume after it executes the function.

Every time the function is invoked, some memory locations will be set apart. If the distance is 1,000 steps, the function will be called 1,000 times, and each time, 1,000 locations will be reserved.

Besides consuming storage space, you should know that each function call consumes some time, utilizing the microprocessor and other overhead, as well.

Ending Recursive Algorithms

Once you grasp the concept, recursive algorithms and recursive function calls are very easy to use. The most important thing to keep in mind when you use a recursive solution is to make sure that the recursion ends at some point. You can proceed with the computation only when a recursive function returns.

In our examples, it was easy to see when the recursion ends. The recursion ends when the robot reaches a wall. But what if there is no wall? We could use this program only because we were guaranteed to find a wall. Likewise, while we explore the maze, either we ask for user input or we use the timer to limit our search.

In any event, make sure you clearly know the condition that signals the end of the recursion, and make sure that it will be reached. Observe also that any variables or objects that are passed by reference or defined globally are not re-created!

The Great Escape—Part 3

Since we can use a recursive function to compute the distance to the next wall, it seems perfectly possible that we can use a recursive function with the `c4search.cpp` program to keep track of how Tracer moves around the maze.

When you compute the distance to the next wall, use a simple recursive algorithm to make sure that Tracer can come back to the original location. If we keep moving left and right to explore the maze, can we still use a recursive function to make Tracer come back? Yes, indeed! All we have to do is to undo the turns and steps.

Let's start with a small modification to `c4search.cpp`. In this new program, instead of using a loop to ask whether she may come back, we use a recursive function to do the same thing. Here is the program `c4back1.cpp`:

```
#include "franca.h"
// Recursive maze exploration
Robot Tracer(1);        // c4back1.cpp
void explore1()
{
 int direction;         // The direction she will face
 if(yesno("Should I go on?"))
 {                      // As long as you answer
   do                   //      yes, this will be done
   {
      direction=ask("which way should I go?");
      Tracer.face(direction);
      Tracer.mark(7);
   }                    // Repeat until the direction
                        //      leads to a free square
   while (Tracer.seewall());
 // Once the square ahead is clear:
 Tracer.mark();         // Mark it green
 Tracer.step();         // Step forward
 explore1();            // Continue
                        // This part will only be executed after
                        //      exploration is done!
 }
}

 void mainprog()
 {
  explore1();
 }
```

This program is very similar to the original program. Besides a few minor changes, the essential difference is that `c4search.cpp` had a loop that was terminated by your answer to the `yesno` function—this program does not seem to have a loop. Does it? Well, it does have a recursive loop!

As long as you reply *yes* to the question, the robot will move into a new square, and then a call to `explore1` will be made. But a call to `explore1` will simply start this function all over again! Isn't this just like a loop? After all, if we reply *yes* 15 times, we invoke `explore1` 15 times. A new variable `distance` is created every time, and a new value is stored in it. Then, the robot marks the next square and moves into it. What is the difference?

The difference is that each time the function is called, a new set of information, including `direction`, is used. The old `direction` is preserved. When you finally reply *no*, the 15th issue of this function will be done, and it will be able to proceed past the `explore1` function call (for the first time). The function will terminate, the content of `distance` will be discarded, and the 14th issue of the function then resumes right after the `explore1` function call. This sequence of events will be repeated until the original issue of the function is also terminated.

Can we use this to make Tracer come back to the original position? Aren't we in the same situation the recursive function was in to measure the distance to the wall?

Indeed, we are. All we have to do is to undo the step taken in the current issue of the function—as in the `findwall` functions. Remember that, this time, we are facing a different direction. But we know the direction that Tracer was facing when we last told her to explore, don't we? It is stored in the variable `direction`. So, if the direction was north, we have to make Tracer face south, and then step. Can you figure out the rest? Please do!

Are You Experienced?

Now you can…

Identify problems that can use recursive solutions

Understand how a recursive algorithm works

Use recursive functions

SKILL

TWELVE

COMPLETING A SHORT PROJECT

- Creating a user interface
- Moving a robot
- Choosing functions
- Implementing a simple application

In Skill 12, you will continue to develop your skills in solving problems and developing applications. You will work on a short project that requires you to design and build software. This project will use `Robot` objects.

Creating the User Interface

In this short project, you will develop a program that can place the robot in any location in an empty room. The location will be determined by specifying the following coordinates:

- Distance, in steps (or squares), from the left wall (horizontal coordinate)
- Distance, in steps (or squares), from the upper wall (vertical coordinate)

The purpose of this software is to keep asking the user for the desired location, and then to move the robot to this location. Since the room has no inside walls, you can rest assured that the robot will not crash, as long as the coordinates remain in the valid range.

For the user interface, use the robot itself and a pair of boxes that indicate the current coordinates (horizontal and vertical). You may consider using the following declarations for these boxes:

```
Box X("Horizontal:"),Y("Vertical:");
```

Moving the Robot—A General Description

The robot always starts in the upper-left square. Since this square is one step from the left wall and one step from the upper wall, it is correct to consider this square to have the coordinates (1,1).

It is important to keep track of the robot's coordinates at all times. If the robot moves one step to the right (east), the horizontal coordinate increases by one. You can determine what happens when the robot moves in other directions.

The software essentially will consist of the following steps:

Repeat:

Get new horizontal and vertical coordinates.
Move robot to new coordinates.
Update current coordinates.

What else do we need to explain? You know how to make a loop, and you will probably have no problem asking for new coordinates to be input. The really interesting problems, then, are to move the robot and to update the coordinates.

What Is Each Move?

Moving from one location to another is really simple. Suppose that the robot is in location (1,1) and that you are requested to move it to (4,9). What do you have to do?

Nothing could be easier:

- Horizontally, you have to move from square 1 to square 4—a total of 3 steps.
- Vertically, you have to move from square 1 to square 9—a total of 8 steps.

So, if the robot is in position (*x*,*y*), and you want to move it to (*newx*,*newy*):

- Horizontally, you have to move *newx–x* steps.
- Vertically, you have to move *newy–y* steps.

Is there any problem with this? No, not really—unless you have to move a negative number of steps, which means moving west (horizontally) or moving north (vertically).

Choosing Functions That Help

At this point, it may be a good idea to design a function to deal with this problem. Suppose that we have a function that tells Tracer how many steps to move horizontally and vertically. This function will check whether the number of steps is positive or negative, and move in the appropriate direction.

```
void move( int xsteps, int ysteps)
```

To make our design simple, we may think of another function that actually moves the robot from a given coordinate to another:

```
void moveto(int &currentx,int &currenty,int newx,int newy)
```

Since we may want this function to automatically update the coordinates, we can pass by reference. If these functions are available, the description of the main body of the loop can be expanded either in algorithmic or program form:

```
// Input new horizontal coordinate:
     horiz=ask("horizontal:");
// Input new vertical coordinate:
     vert=ask("vertical:");
// Move to new location:
     moveto(currentx,currenty,horiz,vert);
// Display coordinates:
     X.say(currentx);
     Y.say(currenty);
```

The loop itself can be controlled very easily:

```
while(yesno("Want to move?"))
```

At this point, your `mainprog` is just about complete. All you need to do is to declare and initialize the variables.

The *moveto()* Function

The code for the two functions, `moveto` and `move`, is interesting. The following function:

```
void moveto(int &currentx,int &currenty,int newx,int newy)
```

is supposed to move the robot from the current coordinates to the new coordinates, and to update the current ones. As we know, this function can use the `move()` function. How can you do this?

As we have seen before, all we have to do is to move the robot *newx–currentx* steps horizontally and *newy–currenty* steps vertically. Or, use the `move()` function as follows:

```
move(newx-currentx,newy-currenty);
currentx=newx;
currenty=newy;
```

It's still easy, isn't it? We left all the trouble for the next function!

The *move()* Function Refined

The `move()` function was mentioned above, but it was not completely explained. Now, we can completely define the following function:

```
void move( int xsteps, int ysteps).
```

This function should verify whether `xsteps` is positive or negative, and then move in the appropriate direction. The same procedure is then applied to `ysteps`.

All you have to do is to turn the robot to face either east (positive *x*) or west (negative *x*), and then step the number of steps that has been specified. Here is how you can do it in the horizontal:

```
void move(int x,int y)
{
  if(x>0)            // Check horizontal direction
  {
     Tracer.face(3);
     Tracer.step(x);
  }
  else
  {
     Tracer.face(1); // If x is negative,
     x=-x;           //      make it positive
     if(x!=0) Tracer.step(x);
  }
  if(y>0)            // Now do the same for y
  {
     Tracer.face(2);
     Tracer.step(y);
  }
  else
  {
     Tracer.face(0);
     y=-y;
     if(y!=0) Tracer.step(y);
  }
  Tracer.face(3);
}
```

Notice that you have to step a positive number, and you cannot step *zero*.

Checking Errors

It is a good idea to avoid, when possible, erroneous user input that causes your program to misbehave. In this case, a range of coordinates is allowed, but the robot must remain in the room. You can restrict the user input of the coordinates to be greater than zero and less than 13.

You can either send an error message or keep asking for the correct input. The latter solution could be as follows:

```
do
    {
       horiz=ask("horizontal (between 1 and 13):");
    } while ((horiz<1) || (horiz >13 ));
```

The Complete Robot Project

Here is a complete listing of the modified `c4move2.cpp` program:

```
#include "franca.h"     // c4move2.cpp
#include "robot.h"
Robot Tracer;
void move(int x,int y)
{
  if(x>0)               // Check horizontal direction
  {
     Tracer.face(3);
     Tracer.step(x);
  }
  else
  {
     Tracer.face(1);    // If x is negative,
     x=-x;              //      make it positive
     if(x!=0) Tracer.step(x);
  }
  if(y>0)
  {
     Tracer.face(2);
     Tracer.step(y);
  }
  else
  {
     Tracer.face(0);
     y=-y;
     if(y!=0) Tracer.step(y);
  }
  Tracer.face(3);
}

void moveto(int &currentx,int &currenty,int x,int y)
{
  int stepx,stepy;
  // How many steps in the horizontal?
  stepx=x-currentx;
  // How many steps in the vertical?
  stepy=y-currenty;
  move(stepx,stepy);
  // What are the new coordinates?
  currentx=currentx+stepx;
  currenty=currenty+stepy;
}
```

```
void mainprog()
{
  int currentx=1,currenty=1;
  Box X("Horizontal:"),Y("Vertical:");
  int horiz,vert;
  for(;yesno("Want to move?");)
  {
     do
     {
        horiz=ask("horizontal (between 1 and 13):");
     }
     while ((horiz<1)||(horiz>13));
     do
     {
        vert=ask("vertical (between 1 and 13):");
     }
     while ((vert<1)||(vert>13));
     moveto(currentx,currenty,horiz,vert);
     X.say(currentx);
     Y.say(currenty);
  }
}
```

Try These for Fun…

- Modify the point-of-sale terminal software to count the following items:

1. The number of sales that had a total value under $10.00.
2. The number of sales that had a total value between $10.00 and $50.00.
3. The number of sales that had a total value between $50.00 and $100.00.
4. The number of sales that had a total value over $100.00.

- This information is to be displayed only after the last customer has checked out.
- Write a program to input grades from the keyboard and to compute the average of all the students who scored 50 or more.
- The factorial function for a nonnegative integer *n* can also be defined recursively:
- If *n* is zero, the factorial is 1.
- If *n* is greater than zero, the factorial is *n* multiplied by the factorial *n–1*.
- Write a function to compute the factorial that uses this recursive definition. Test your function.
- Write a function to compute and return the absolute value of a floating point number. Test your function.

If *y* is positive, the result is *y*. If *y* is negative, the result is *–y*.

- Write a program to request that grades (between zero and 100) are input from the keyboard. This program should compute how many of these grades are between 0 and 25, how many of these grades are between 26 and 50, how many of these grades are between 51 and 75, and how many of these grades are between 76 and 100. Display the results.

Are You Experienced?

Now you can…

Analyze and decide on a user interface

Write a general description of an application

Choose functions and program pieces that can be useful in your application

Implement a simple application