PART III

USING REPETITIONS

Computers are great at repetitive work. In fact, their usefulness is so extensive because most problems involve an incredible amount of repetitions.

You will now learn how to specify that a piece of program is to be repeated, how to specify how many times it should be repeated, and how to find out the results that will be generated each time.

These pieces of program are called loops. You will learn about how a loop works and what the statements are that can be used to build them. You will then learn how to design a loop in order to solve your problems. Finally, you will start working on a down-to-earth application that develops a simplified point-of-sale terminal.

SKILL

SEVEN

COUNTING AND REPEATING

- Repeating with *while* loops
- Repeating with *do/while* loops
- Repeating with *for* loops
- Using conditional repetitions
- Nesting loops

In a real fitness class, Sal must perform each exercise several times. For example, he has to repeat the jumping jack a certain number of times. This is the simplest kind of repetition: a given set of instructions is to be repeated a predetermined number of times. If you were Sal's instructor, you would probably instruct him to repeat a jumping jack 15 times.

For a smart guy like Sal, this would probably suffice. He would understand the meaning of the word *repeat*, and he would know what you mean by a jumping jack. Finally, *15 times* complements the order to repeat and, therefore, the jumping jack will be repeated 15 times.

Unfortunately, computers are not as smart as Sal and we must make sure they understand exactly what we want them to do. The process is easy to understand if you think of how a real person executes the instruction to repeat a jumping jack 15 times.

If you want to repeat something, you have to count how many times you have already repeated and stop when you reach the desired count. Very likely, you will see people exercising and shouting, "One! Two! Three!" The process we follow is illustrated in Figure 7.1.

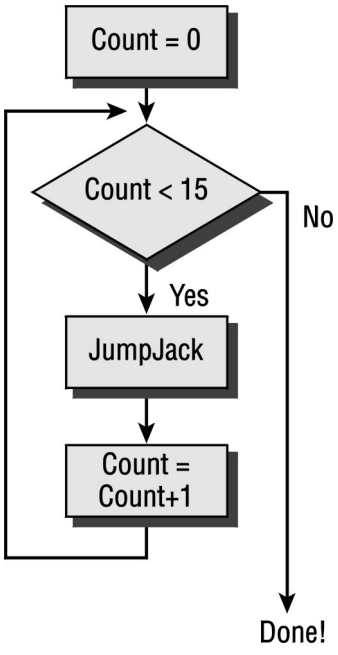

Fig 7.1 Counting and Repeating

Just like humans, the computer also needs to store a count that will go *one, two, three.* To do this, we can use an integer variable. This skill will show you how to use integer variables to include counting and repeating in your programs.

Understanding Simple Repetitions

We are now ready to look at simple repetitions that cause a piece of program to repeat a specified number of times. There are three possibilities:

- *while* loops
- *do/while* loops
- *for* loops

Repeating with *while* Loops

The repetition illustrated in Figure 7.1 can be explained as follows:

Set count to zero.
While count is less than 15, repeat the following:
    JumpJack.
    increment count.
You are done!

in which *count* is merely a variable that keeps the count of how many jumping jacks have been completed. At the start, you can make this equal to zero, since you haven't done anything so far.

The next line, *while…*, explains that the indented sequence that follows is to be repeated while the condition *count is less than 15* is true.

Isn't this what you actually do in your mind? And why is the sequence indented? We will find out a little later.

Loops Defined

Figure 7.1 illustrates the sequence in which we want the instructions to be executed. This is called a *flowchart*. If you start at the top (count=0) and follow the indications in the flowchart, you should observe that the next thing to be done is to compare the count with 15. There are two alternate directions to take, according to the result. Since the count is zero initially, the next steps will be to perform a JumpJack and then increment the count by one.

After that action is completed, the arrows indicate that we should go back to comparing the count with 15. Well, the count is now 1, which is still less than 15, so the same thing is repeated. The path followed in the flowchart each time we repeat this sequence is a closed path that resembles a loop. For this reason, programmers call a piece of program that gets repeated a *loop*.

This loop is executed 15 times (since the count is zero until it hits 14, both inclusive). When the count hits 14, the program performs a JumpJack and then increments the count, resulting in the new value, 15. Now, when the comparison takes place, the count is compared with 15—this time it is no longer less! Finally, the alternate path is taken and we leave the loop.

Infinite Loops

Suppose that we did not include the following statement:

```
count=count+1;
```

to update the count in the program. When would we leave the loop? Never!

We would keep comparing the count (which would always be zero) with 15, and the count would always be less than 15 (since the original value wouldn't change). This would result in a well-known bug called an *infinite loop*. This example is very simple and, hopefully, you can see the problem. In other cases, the problem may be more complex, and it may not be so easy to notice that the loop is infinite.

Don't worry, you will eventually make some of your own infinite loops…

If you suspect you are executing a program that is in an infinite loop, you can cancel the execution by pressing Ctrl+Alt+Delete (at the same time). While you are in Windows, this will bring a message to the screen that allows you to cancel the current program by pressing Enter. The program execution will be canceled, and you will go back to the C++ compiler.

Compound Statements

Take another look at the piece of program we are examining. Why is it that the steps:

JumpJack
increment count

are indented?

You must clearly indicate which set of instructions is going to be repeated. For example, you don't want to repeat *You are done!* every time, do you? As a matter of fact, when you want to group some statements to be treated as a single, big statement, you enclose them in braces to denote that those statements be treated as one compound statement.

A *compound statement* is a group of statements (enclosed in braces) that is to be treated as one statement for purposes such as repetitions. (It is also common to refer to compound statements as *blocks*.) When describing the algorithm, you may simply indent this group of statements. In C++, you actually have to use braces.

All functions, including `mainprog`, contain a compound statement (which is why a semicolon is not used).

Don't place a semicolon between the `while` statement and the compound statement! The semicolon ends a statement right where it is placed.

For example:

```
while (count<15);      // This semicolon is wrong!
{
   JumpJack();
   count=count+1;
}
```

will cause the computer to understand that while `count` is less than 15, you want to repeat *nothing*. Does this look like an infinite loop? Well, it may not look like one, but it certainly is! The empty loop will be repeated forever, because the value of `count` is never incremented.

Also, notice that in the correct program, every time you perform a `JumpJack`, you increment `count`. This means that `count` will equal 1, 2, 3…14. After `count` equals 14, the condition is still good (after all, 14 is less than 15). When we repeat the sequence one more time (the 15th), `count` will be incremented to 15.

At this point, the condition is checked again before the repetition takes place. Since `count` (the number of times we have repeated) is now 15, the condition fails (15 is not less than 15) and the repetition stops! After the loop completes, the program resumes with the next instruction (after the closing brace).

You could start with `count=1` and check for a count that is less than or equal to 15, or less than 16. (*Less than or equal to 15* and *less than 16* amount to the same thing.)

In C++, the repetition with which we have been dealing could be written as follows:

```
count=0;             // Set initial count to zero
while (count<15)     // Now repeat the sequence below
{                    //     fifteen times.
     JumpJack(Sal); //          do a JumpJack
     count++;       //          update count
}                    // Sequence to repeat ends here!
```

The process followed in the `while` is illustrated in Figure 7.1.

The program above could also be written as follows:

```
count=0;
while (count<15)
{JumpJack(Sal);
count++;}
```

Indentation makes the program more readable.

Using Operators in the *while* Loop

The keyword `while` indicates that a sequence is to be repeated while the condition is true. The condition is shown inside parentheses and reflects the relationship between two values. In this case, we are interested in a count that is less than 15; therefore, we used the operator < (less than). Other operators are possible:

==	equal to (notice that there are two equal signs)
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	not equal to

Conditions

When you want to repeat a sequence of statements, you must specify when you want the sequence to start repeating and when you want it to stop repeating. To do this, specify the condition for executing the sequence. This condition is usually represented by a relational expression. The relational expression is shown right after the keyword `while` and is enclosed inside parentheses.

In this example, the relational expression is the following:

```
count < 15
```

which means that we want to repeat the sequence as long as the value of `count` is less than 15. The symbol for less than (<) is a relational operator, because it expresses the relationship between

objects. Of course, you could use the other operators, as well. There may be times when you have to specify a condition that is much more complex. In SKILL ten, we will learn how to combine them into more complex relational expressions.

Try These for Fun…

- Write a program to make Sal perform five jumping jacks and five leftrights.
- Write a program to make Sal perform a jumping jack followed by a leftright five times.
- Write a program to make Sal perform a jumping jack five times and to then make Sally perform a jumping jack five times.
- Write a program to make both Sal and Sally perform a jumping jack together (simultaneously) five times.
- Write a function, `JmpJack`, that takes two parameters:

  ```
  void JmpJack( athlete somebody, int howmany)
  ```

  - This function should make the athlete `somebody` repeat a jumping jack for the specified number of times.

Repeating with *do/while* Loops

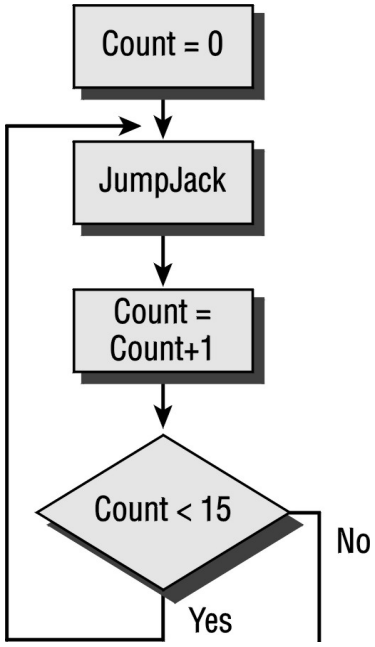

Fig 7.2 The do/while loop

The `do/while` statement is a slightly different variation of the `while` repetition. In the `while` loop, the condition is tested before you start the loop. As a result, if the condition is not true at the beginning, the loop is not executed even once!

`do/while` makes the comparison at the end of the loop, as shown in Figure 7.2. No matter what happens to the condition, the loop is executed at least once. After this first execution, the condition is checked and then the loop may or not be repeated.

In `do/while`, the keyword `do` indicates the beginning of the loop. The keyword `while`, which is followed by the condition, indicates the end of the loop. Notice the semicolon after the condition!

For example, the same effect could be achieved by the following program:

```
count=0:
do
{

JumpJack(Sal);
count++;
}
while (count<15);
```

Notice that the actual loop is enclosed in braces and the `while` is located after the braces.

Repeating with *for* Loops

You may often find the `for` statement to be easier and more convenient than `while`. `while` requires you to initialize the count and to update its value. If you forget either one of these two things, you will surely end up in trouble.

The following program:

```
count=0;
      while (count<15)
      {
       JumpJack(Sal);
       count++;
      }
```

could also be written as follows:

```
for (count=0;count<15;count++)
      {
       JumpJack(Sal);
      }
```

The two structures are very similar, except that `for` allows you to specify the initial value, the condition, and the update all in the same place! Inside the parentheses, there are three parts separated by semicolons:

- The first part, `count=0`, is an expression that will be executed only once before the repetition takes place. This is useful to set the initial value for the count we are using.
- The second part, `count<15`, specifies the condition that keeps the repetition going. Just like in `while`, this condition is checked and the repetition takes place if it is true.
- The third part, `count++`, is an expression that will be executed once at the end of each repetition. It is useful for specifying how we want to update the count.

You might instead use the following statement to achieve the same results:

```
for (count=1; count<=15; count++)
```

It may be more natural to start counting from one and to stop at the desired count than to start counting from zero.

> & Remember, the expressions are separated by semicolons!

It is also possible to declare a new variable inside `for`. For example:

```
for(int counter=1;counter<=15;counter++);
```

In the example that follows using the program `c3askfor.cpp`, we use `for` to repeat, and we request that the number of times be input through the keyboard using the `ask` function.

```
// Using user input...          // c3askfor.cpp
// This program uses input
// Revised September, 7th, 1994.
#include "franca.h"
void JumpJack(athlete she)
{
   // We make "she" perform a JumpJack
   she.up();
   she.ready();
}
void mainprog()
{
   athlete Kim;
   int howmany,done;
   // Ask how many times you want her exercising
   howmany=Kim.ask("How many jumping Jacks?");
   // Now repeat as many as told:
   for(done=1;done<=howmany;done++)
   {
       JumpJack(Kim);         // Do a JumpJack
       Kim.say(done);
   }
       Kim.say("done");       // See the quotes?
}
```

In this program, we don't know how many times the JumpJack will be repeated while we write the program. We simply set apart a variable, `howmany`, and tell the computer to execute the loop while `done` is less than or equal to `howmany`.

THE EQUIVALENCE OF `FOR` AND `WHILE`

A piece of program that includes `for` is entirely equivalent to one using `while`. The initial expression in `for` is executed before `while`, and the final expression is executed just before the end of the `while` loop.

Thus, a sequence such as the following:

```
for( <exp1> ; <exp2> ; <exp3> )

{

   <statements>

}
```

is the same as:

```
<exp1>;

while(<exp2>)

{

   <statements>

   <exp3>;

}
```

Using Conditional Repetitions

In the previous section, we examined the simplest kind of repetition, in which we knew exactly how many times we wanted to repeat the sequence.

We will now examine the other kind of repetition, in which the number of times is not known beforehand. Of course, you may come up with some examples in real life in which you have to repeat something an unknown number of times until, for some reason, you know you are done.

The only difference between these repetitions is the condition that you have to test. Previously, you tested if the count was still less than or equal to the number of times you wanted to repeat. Now, we aren't going to pay attention to the number of times we want to repeat. Instead, we'll try to check whether we are already done.

Checking Input Values

An interesting and useful application of repetitions is to verify input data. Suppose you want the user to type a value, and you want to make sure it is not negative. This could be solved with the sequence below:

```
float value;
value=ask("Enter a positive value:");
while ( value < 0 )
{
   value=ask("Wrong, enter a POSITIVE value:");
}
```

This piece of program will keep asking the user to reenter the value as long as the input is negative. This kind of data validation is very important, and you should include similar precautions in your programs. You can never trust what the user will input, but you can always trust what your program will do!

Example—At the Store

Suppose we want to compute the change for merchandise sales at a store. The program `c3store1.cpp` illustrates a solution to this.

```
#include "franca.h"
void mainprog()              // c3store1.cpp
{
  float price,change,amount,tax=8.25;
  price=ask("Input the price:");
  price=price+price*tax/100;
  Cout<<price;

  amount=ask("Enter the amount:");
  while (amount<price)
  {
     amount=ask("Not enough, re-enter amount:");
  }
  change=amount-price;
  Cout<<change;
}
```

After the amount is first input, a `while` loop will be repeated as long as the amount is less than the price. If the amount is sufficient, this loop will not be executed (not even once). We may proceed to compute the change. However, if the amount is not sufficient, the loop will be

entered to request that a new value be input. It is nice that the program will not go on unless the correct amount is entered, no matter how many times the user makes a mistake!

An equivalent solution to this problem could make use of `do/while`. Since this kind of loop tests the condition only after the loop has been executed, the resulting program can do without one of the inputs. In this case, the `while` loop could be substituted for the following:

```
do
{
   amount=ask("Enter amount:");
}
while (amount<price);
```

Notice how `do/while` served this case better—fewer lines of code were needed. In this case, this solution has the inconvenience of failing to report what is wrong if the user keeps typing an insufficient amount.

Checking the Time

There may be several other reasons that determine the end of the loop besides the number of repetitions. As mentioned, checking the correct value of some input data is one of them. Another interesting case is when we use the time to determine how long a loop should be performed.

As you may have figured out, all you have to do is to write the correct condition in `while`, `do/while`, or `for`.

Example—At the Fitness Class

Let's suppose that Kim wants to exercise for 15 seconds as an example of conditional repetition controlled by time. As you can guess, we cannot tell in advance how many exercises will be done in 15 seconds unless, of course, we know the precise duration of each exercise. This is actually the case in real life. There are many events that you keep doing until the time is up, as in the example program `c3dotime.cpp`, shown below.

```
// Testing repetitions with time... c3dotime.cpp
// Revised August/8/94
#include "franca.h"
void JumpJack(athlete she)      // Function JumpJack
{
   she.ready();
   she.up();
   she.ready();
}

void mainprog()                 // Main Program
{
   athlete Kim;                 // Declare Kim as an athlete
   int howmany=1;               // Declare a counter
   Clock timex;                 // Create a clock
   while (timex.time()<12)      // While time is less than 12
   {
       JumpJack(Kim);           // JumpJack
       Kim.say(howmany++);      // Say how many times
    }
}
```

Execute this program. Notice that the condition in `while` simply checks the time in the clock and keeps repeating while the time is less than 12.

Example—How Fast Is Your Computer?

It is an interesting exercise to see how fast Kim can exercise on your computer. If you modify the program above by inserting a zero as an argument to the messages `ready` and `up`, these movements will be done as fast as possible. See how many exercises can be done in 12 seconds, for example. The result will vary according to the type of computer that you are using. You may try to check your results against your colleagues' results. The faster the computer, the more exercises that can be done.

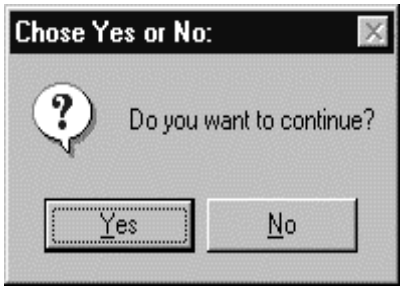

Fig 7.3 The yesno box

Using the *yesno* Function

In some situations, you may want your program to ask the user if a loop is to be repeated. Take the `c3store1.cpp` program, for example. In this program, we requested some input regarding the price and amount tendered to compute the change.

Very likely, this task is going to be repeated many times during the day. We don't want to tell the computer to run our program every time a customer wants to buy something. The appropriate solution is to keep this program running and asking for input. But, when should we stop? Well, we could include a question such as *Do you want to continue? Yes or No?* at the end of the loop. If the user answers *yes*, the loop will be repeated.

How do we know whether the user answers *yes* or *no*?

To make our life simpler, `franca.h` includes a function that displays a dialog box, in which the user can click a box marked *Yes* or a box marked *No*. We can pass a sentence as an argument to this function, and this sentence will also be displayed on the screen. Figure 7.3 illustrates the dialog box generated by `yesno`.

The dialog box results from the following call to `yesno`:

```
yesno("Do you want to continue?");
```

The `yesno` function also returns a value, which makes it possible to use the function call directly in an expression. If the user clicks Yes, the returned value will be one; otherwise, it will be zero. A revised code for the program is presented below as program `c3store2.cpp`.

Essentially, we included the body of the previous program inside a `do/while` loop. The `yesno` function is invoked directly as the expression for the loop.

```
#include "franca.h"
void mainprog()                          // c3store2.cpp
{
  float price,change,amount,tax=8.25;
  do
  {
   price=ask("Input the price:");
   price=price+price*tax/100;
   Cout<<price;

   amount=ask("Enter the amount:");
   while (amount<price)
   {
      amount=ask("Not enough, re-enter amount:");
   }
   change=amount-price;
   Cout<<change;
  }
  while (yesno("Do you want to continue?"));
}
```

This function may be used in any expression and is not restricted for use with a `do/while`.

> & Choose your question so that most of the time the answer is *yes*. This is because the default is *yes*, and the user may just hit Enter instead of moving the mouse to click No.

> & The `yesno` function is not available in standard C++—it is only available with the software developed for this book.

Nesting

You are often faced with the situation in which you have to repeat some directions that include another repetition:

Take two capsules a day for 10 days.

You have to repeat an action 10 times that is a repetition itself:

Repeat the following 10 times:
    Repeat the following 2 times:
    swallow a capsule.
Wait until the next day.

You will be swallowing a total of 20 capsules.

Notice that one repetition is "nested" inside the other. You h
inner repetition to proceed with the outer one. Figure 7.4 shows what
loop that is repeated every day. Figure 7.5 shows the complete proces
there is an *inner* and an *outer* loop.

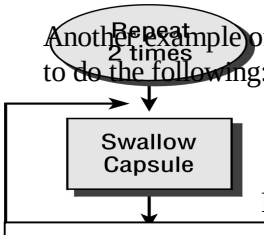

Fig 7.4 Every day loop

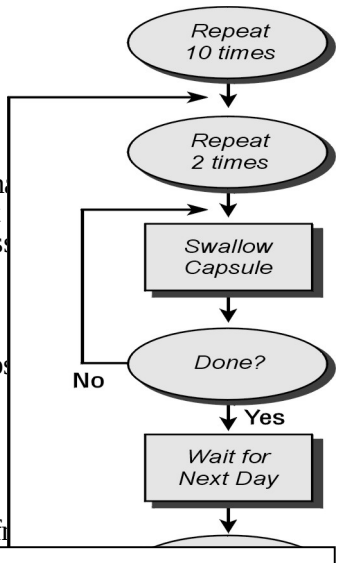

Fig 7.5 What you do in 10 days

Another example of nesting appears in the fitness class. Suppos
to do the following:

Repeat this 5 times:
    Repeat this 10 times:
    jumping jack.
    Repeat this 5 times:
    leftright.
Repeat this 3 times:
    jumping jack.

You are to do 10 jumping jacks followed by 5 leftrights, and then repeat this sequence 5 times (a total of 50 jumping jacks and 25 leftrights). Then, you do an additional 3 jumping jacks.

The first set of jumping jacks is nested inside another repetition. The second set is not. In C++, this program could be written as follows:

```
#include "franca.h"
void mainprog()
{
  athlete Sal;
  int bigcount,smallcount;      // To keep the count…

  for (bigcount=1;bigcount<=5;bigcount++) // repeat 5
  {
     for (smallcount=1;smallcount<=10;smallcount++)
     {
       JmpJack(Sal);
     }
     for (smallcount=1;smallcount<=5;smallcount++)
     {
       leftright(Sal);
     }
  }                             // End of 5 times
  for (smallcount=1;smallcount<=3;smallcount++)
  {
      JumpJack(Sal);
  }
}
```

Practicing Repetitions with Tracer

The robot operations can now be recoded using repetitions. You can code a function to draw a line of any size specified. Simply include a parameter that lists the number of steps required to draw that line. It is still possible, and desirable, to keep the ability to draw the line in any given color, as dictated by a second parameter.

Specify what the header of this new function will look like:

```
void line (int size, int color = 2)
```

As we have seen before, this function will have to repeat the following statements:

```
mark(color);
step();
```

for as many steps needed. Thus, if the parameter size contains the size of the line in number of steps, this sequence should be repeated exactly that number of steps, minus one (remember the first square?).

You may find it easy to implement this by using `for`:

```
for ( int howmany=1;howmany<size;howmany++)
```

By specifying the condition `howmany<size`, you will be executing the sentence *size–1* times.

A complete listing for this function could be as follows:

```
void line (int size,int color=2)
{
  for(int howmany=1;howmany<size;howmany++)
  {
     Tracer.mark(color);
     Tracer.step();
  }
}
```

Before you proceed any further, notice that this function will not require any change to the code no matter how many steps you want. Everything is determined by the parameter size. The program `c3square.cpp` below contains the complete code, with the other functions modified. This program draws two squares of different sizes and colors.

```
#include "franca.h"    // c3square.cpp
Robot Tracer;
void line (int size,int color=2)
{
  for(int howmany=1;howmany<size;howmany++)
  {
     Tracer.mark(color);
     Tracer.step();
  }
}
```

A repetition could simplify the square function itself :

```
void square(int size,int color=2)
{
  for(int turn=1;turn<=4;turn++)
  {
     line(size,color);
     Tracer.right();
  }
}
```

We could still use the same `mainprog`:

```
void mainprog()
{
  Tracer.face(3);
  square(3);
  Tracer.step(5);
  square(4,5);
}
```

Exploring a Room

Tell Tracer to walk along all the walls of a rectangular room and come back to the original position. How can you solve this problem? Some preliminaries follow.

You can now take advantage of the fact that Tracer can detect whether there is a wall ahead by sending her the following message:

```
Tracer.seewall();
```

Or better still, you can have her keep moving while the following condition:

```
Tracer.seenowall();
```

is true.

This means you can keep her walking until she finds a wall, without actually knowing how many steps are needed! As long as the condition `Tracer.seenowall()` is true, she can keep walking. Do you know how to do this? Try the following:

```
for(;Tracer.seenowall();)
{
   // You might also mark: Tracer.mark();
   Tracer.step();
}
```

What happened to the other two expressions inside `for`? They vanished! You don't really need the other expressions in this case. There is nothing to initialize and nothing to be done at the end of each iteration.

If you want to count the steps until she reaches the wall, you might then modify the code as follows:

```
for(int howmany=0;Tracer.seenowall;howmany++)
{
   // You might also use mark: Tracer.mark();
   Tracer.step();
}
```

A Problem to Solve

Suppose that we have an empty rectangular room and that Tracer is in a corner, facing east. This is illustrated in Figure 7.6.

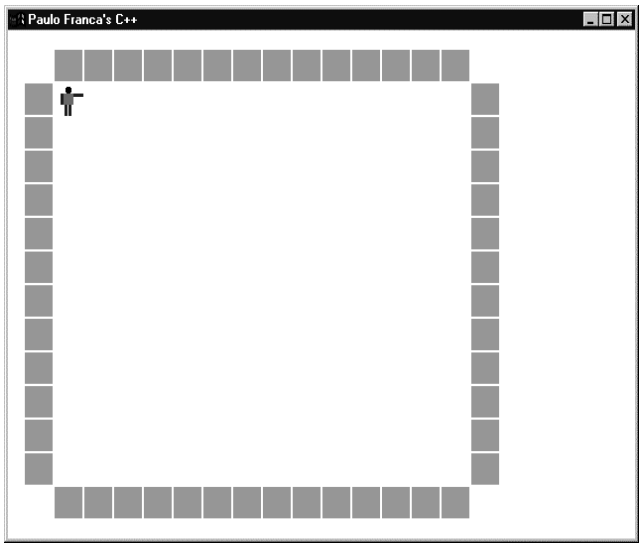

Fig 7.6 Tracer facing east

Can you figure out how to have Tracer move around the room? Since you know that the room is rectangular, you may want to tell Tracer to do the following:

- Step ahead until she finds a wall. Once the first wall is found, there are only three left!
- Have her turn and step ahead until she finds another wall. There are only two left now! What do you do now?

Do you think you can complete this by yourself? Why not try it? After you try, you can check the solutions shown below.

Solution 1: Walk around four walls.

This program makes Tracer move around, one wall at a time. As you can see, we have her step until she finds a wall, turn right, step until she finds another wall, turn right, etc.

The following is a possible algorithm:

Step ahead until you find a wall;
turn right.
Step ahead until you find a wall;
turn right.
Step ahead until you find a wall;
turn right.
Step ahead until you find a wall;
turn right.

Solution 1 is correct, but, as a programmer, you should not be happy with it. Two statements are written four times in exactly in the same way! This is an obvious repetition.

> & Thou shalt not write the same line of code twice!

Solution 2: Repeat four times.

A better solution is to group the actions required to find each wall, and then to repeat them for each wall. The algorithm can be as follows:

Repeat the statements below four times:
    Step ahead until you find a wall;
    turn right.

This is a good solution to this problem. Solution 3, discussed below, is not necessarily better.

Solution 3: Use a function.

A third possibility would be to use functions. For example:

```
void findwall();
```

makes Tracer move until she finds a wall and then makes her turn right. If this function is made available, the algorithm would become as follows:

Repeat the statement below four times:
    findwall.

It is barely justified to use a function in this case, because the procedure is so simple.

Solution 4: Use a function that returns a value.

Finally, let me comment on the possibility of using a function that returns a value—the number of squares in which she stepped until the wall was found.

Here is one implementation of the function:

```
int findwall()
{
  int steps=0;
  while (tracer.seenowall())
  {
     tracer.step();
     steps++;
  }
  tracer.right();
  return steps;
}
```

This function adds the convenience of counting the steps. Although this computation is not necessary at the moment, it may be useful sometime in the future. For example, if we want to move Tracer and keep track of how many steps she takes, we could do the following:

```
int howfar;
howfar=findwall();
```

Although the function returns a value, you don't have to use it. This function could be used just like the previous version was used:

```
for( int i=1;i<4;i++)
       findwall();
```

Try These for Fun…

- Write a function:

  ```
  int distance()
  ```

  - This function should measure the number of steps needed to reach the wall Tracer is facing. However, Tracer should measure the distance, then come back to the original location and face the same direction.
- Suppose that Tracer is inside a rectangular room. You don't know where she is, nor the direction in which she is facing. Have her measure the room.
- Suppose that Tracer is inside a rectangular room. You don't know where she is, nor the direction in which she is facing. Have her walk in all the locations in the room.

Are You Experienced?

Now you can…

Write a piece of code that will be repeated

Use *while*, *do/while* and *for* loops

Specify appropriate conditions for loops

Nest one repetition inside another

SKILL

EIGHT

DESIGNING BASIC LOOPS

- Repeating with variations
- Designing loops—Part 1
- Summarizing rules and regulations

Repetition is not necessarily monotonous. In fact, most of the great harmony of nature is caused by patterns that are repeated, but are slightly different from each other. You may ask, How can something be repeated and yet be different?

Well, you do this all the time. As you go through your daily routine, you wake up, get dressed, eat breakfast… It's all the same every day, but:

- You don't wear the same clothes every day, do you?
- You don't eat exactly the same breakfast every day, do you?

So, although you repeat the major actions, some of the details are changed.

Repeating, but Changing

If you look at Figures 8.1 and 8.2, you will find several sets, each consisting of three squares. In each set, you can see that squares are repeated, but the result is far from being the same!

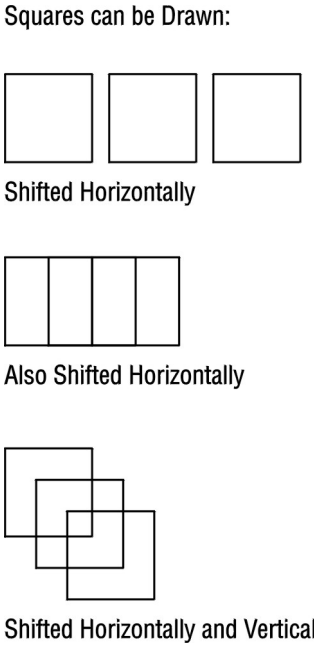

Fig 8.1 Squares of the same size

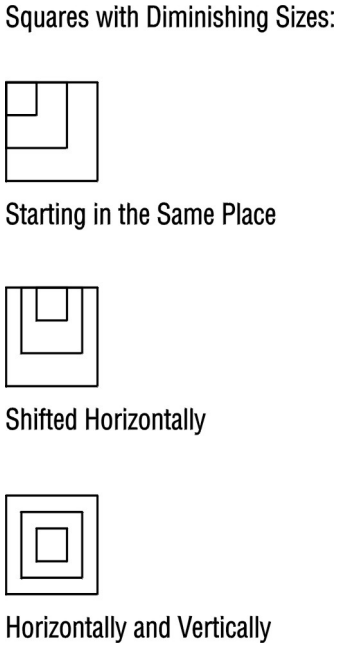

Fig 8.2 Squares with diminishing sizes

If you had to give instructions on how to draw these pictures, you would probably notice that you have to repeat the drawing of a square three times. Each time, the square is a different size and is located in a different place.

If you know how to draw a square of a given size at a given location, you should be able to have Tracer draw any of these pictures for you. Let's agree that the largest square will be of size 12, and the others will be of sizes 8 and 4. Why not use your smart robot to draw these squares?

I can give you a hint for one of the pictures:

- Consider the top set of squares shown in Figure 8.2. All the squares start at the same corner, but their sizes decrease—from 12 to 8 to 4! In other words, you start with size 12, but each time you draw the next square, you decrease the size by 4. Get it?

Let's see how we can do this.

Example—Drawing Squares of Diminishing Sizes

The following is our first attempt in plain English:

Draw a square of size 12.
Draw a square of size 8.
Draw a square of size 4.

By now, you know you can use a function to draw a square. Don't write a new one! Reuse! Once you have this function, you could write the program as follows.

The First Program

Here's what our first attempt would look like:

```
#include "franca.h"
Robot Tracer;
void mainprog()
{
     square(12);
     square(8);
     square(4);
}
```

This program is correct and should work right if you try it—but you should not be happy with it! Why not?

Thou shalt not write the same line of code twice.

Do you see that this program just repeats the drawing of a square? This is repeated three times! A good programmer identifies repeated pieces of code and organizes them, so that they do not have to change if you need to repeat them a different number of times. For example, whether you want to draw 3, 15, or 100 squares, the code will be essentially the same. Ideally, you should obtain something such as the following.

The Second Program (Incorrect)

Here's what our second attempt might look like:

```
#include "franca.h"
Robot Tracer;
void mainprog()
{
     int many,size;
     size=12;
     for (many=1; many <=3;many++)
     {
        square (size);
     }
}
```

Well, this would be correct and very easy if all the squares were the same size, but they are not. The squares shrink by 4 units each time. Wait a moment! If you knew how to compute the next size, you could simply compute the new size every time!

The Third Program

Here's what our third attempt might look like:

```
#include "franca.h"   // c3sq3.cpp
Robot Tracer;
void mainprog()
{
     int many,size;
     size=12;
     for (many=1;many <= 3 ;many++)
     {
        square(size);
        size=size-4;
     }
}
```

Can you understand what happens with the size? Initially, there is a value of 12. Then, you start the repetition and draw the first square with this size (12). Next, you compute a new value for the size. What is it? You simply take the old value and decrease it by 4—this becomes the new value. The next time you draw a square, it will be of size 8. This process will be repeated as the squares keep shrinking.

In some of the other pictures, in which each square starts at a different location, all you have to do is to move Tracer to the new starting point. Do you think the directions to move to a new starting point will be very distinct from each other? I don't think so. If you look at most of the pictures, you will find that once you draw a square, you simply move a couple steps horizontally or vertically (or both) and then you are ready for the next square! It's easy, isn't it?

The complete code can be found in `c3sq3.cpp`.

Repeating the Same Lines of Code

If you compare the first program with the third, you will notice that they both draw the same picture! However, I suggested that you avoid using the first and that you prefer the third. You may be wondering about my opinion. The first program might be easier to understand than the third, yet I argue that the third is preferable. Where is the improvement?

As you compare programs, you may feel tempted to say that the first try is just about as good as the last. Their size isn't much different, and the first one is quite easy to understand. I cannot properly say that you are wrong. In fact, the simple structure of the first program is tempting when it comes to drawing three squares. The problem is that very often you will be faced with larger numbers. If you want to draw 10 squares, you have to add several statements to your first program. How many would you have to add to the last one? None! All you would have to do is modify the initial size and the number of repetitions.

Another valuable benefit of using repetition is when you have to deal with an unknown number of repetitions. The first program will only draw exactly three squares. The last one allows you to draw whatever number is specified in the repetition. If, instead of specifying the number 3, you specified your repetition as follows:

```
for (many=1; many <= total; many++)
```

the sequence would be repeated as many times as the value stored in `total`.

Recognizing Opportunities for Repetition

The role of repetitions should never be underestimated. The more you can identify repetitions in your program, the better. Sometimes the repetition is subtle, making it hard to find what changes from one step to the next. As you develop more experience, you will be able to perceive such changes more easily.

The programs to draw three squares could also utilize nested repetitions. For each square, we have to draw four lines. Then, we have to draw three squares, making a total of 12 lines. Therefore, we could also write the program as follows:

Set size=12.
Repeat 3 times:
    Repeat 4 times:
        Draw a line (size).
        Turn right.
        Compute new size=size–4.

The Fourth Program

In C++, the program described above would be as follows:

```
#include "franca.h"
     int size,squarecount, linecount;
     size=12;
     for(squarecount=1;squarecount<=3;squarecount++)
     {
          for(linecount=1;linecount<=4;linecount++)
          {
               line(size);
               Tracer.right();
          }
          size=size-4;
     }
```

Again, the result is the same as before. In the previous version of the program, we used the function `square` to make the inner repetition invisible. As a result, I hope you agree that the third program is a little easier to understand. However, you may not find it convenient to create functions for everything, and you should become acquainted with nested repetitions, as well.

Try These for Fun…

- Write a program to draw the picture shown in the middle set of squares in Figure 8.2.
- If you'd like to try something more difficult, write a program to draw the picture shown in the bottom set of squares in Figure 8.2.

Designing Your Own Loops—Part 1

Computers are successful only because most problems involve repetition. The procedure to process a customer's check is about the same for all customers and is repeated thousands of times every day. Procedures that compute complex scientific results also involve millions of repetitions of the same piece of program.

You'll learn more about designing loops in "Designing Your Own Loops—Part 2" in SKILL ten.

By now, you possess the means to control the repetition of a piece of a program. The next task is to make sure you know how to use them. There may be situations in which you know something has to be repeated, but you don't know exactly what.

General Guidelines

When a repetition is involved, you have to determine the following:

- Which task is supposed to be repeated?
- What changes are to be performed after each iteration?
- When do you stop repeating?
- What initial conditions are there for the loop?

Which Task to Repeat

I suggest you start by trying to determine which task must be repeated. It may help if you consider the simplified case in which only one repetition is needed. It is also very helpful to think of which task will be completed after each iteration is completed. You may even try to formulate a sentence that remains true every time a loop iteration is finished.

A sentence that remains true every time a loop iteration is finished is called a *loop invariant*.

For example, in the program in which we wanted to draw the diminishing squares (`c3sq3.cpp`) , we wanted to repeat the drawing of a square:

```
square (12 );
```

This would suffice if we had only one square to draw.

However, the square we want to draw will have a different size each time. Therefore, this size is part of what is changing after each loop, because after each square is drawn, this size is changed for the next drawing. Notice that after each iteration, one square is completely drawn. Therefore, the following assertion:

Square number *n* is drawn.

is true for each iteration.

What to Change

Next, you can try to determine what changes from one iteration to the next.

In the example above, only the size of the square changes. Squares should become 4 units smaller each time. This update can be done inside the loop itself or as part of the third expression of a `for`. Most likely, you will also need to count how many iterations have been completed.

When to Stop

Make sure that your loop stops—either by counting iterations or by some other process. In SKILL ten , you will see how to interrupt a loop that is checking some condition. In any event, it is important that you make sure this condition that you are checking to interrupt the loop will happen sometime.

In this example, we want to draw three squares. The loop will stop at the count of three.

How Does the Loop Begin?

Make sure that all the initial values for when the loop begins are correct. You may have to start the count from zero or one, reset a total to zero, or set an initial value for other variables.

Sometimes it is easier to determine the initial conditions after you have designed the loop. In the example, set the initial square size to 12 and the square count to 1.

A Design Example

Suppose that your instructor asks you to assist in computing the average score for all the grades in your class. How would you solve this problem?

You know the average can be obtained by adding all the grades together and dividing the total by the number of students in class. You may have developed the following general explanation for your problem:

Add all grades together.

Average=total divided by number of students.

Write the average.

The real problem now becomes, How do you add all the grades together?

Clearly, you have to read them so that they can be added. You can do this in the same way that you would if you used a calculator:

Repeat while there are grades:

Type a value.

Hit the + key.

Since the + key on a calculator adds the current number to the accumulator, this is the same as the following:

Repeat while there are grades:

Input grade.

Add to total.

If you have any trouble understanding this concept for calculating, you can also try to write a program to compute the average of one, two, or three grades, and transform it into a loop.

At this point, you know what is to be repeated:

Input a grade.

Add to total.

This becomes the following in C++:

```
grade=ask ("Please enter next grade:");
total=total+grade;
```

You can now check what changes are needed from one iteration to the next. In this case, a new grade is input in each iteration and the total is already updated. No more changes are necessary.

When to Stop the Loop

You have to decide which mechanism to use to stop the loop. The easiest way would be to know in advance how many grades there are in the list and to read exactly that many. For example, if there were 26 grades, just use a loop such as the following:

```
for (int which=0;which<26;which++)
```

Will this work? Sure it will. However, it is not very flexible. Next month, the instructor will come back to you because the class now has only 24 students, which will require program changes.

There are a few alternatives to this structure:

- You can have the program ask for the number of students in class.
- You can check whether the instructor wants to input more grades after each input.
- You can establish a sentinel value that will signal when no more data are available.

The first alternative is easy to implement—just include some statements before the loop:

```
int class_size;
class_size=ask("enter the number of grades:");
```

and change the `for` statement to the following:

```
for (int which=0;which<class_size;which++)
```

This is a reasonable solution, but it does require that someone counts how many grades there are before inputting them. The user may not like this if the number is large.

The second alternative can also be easily implemented by means of the `yesno` function. Remember that you can ask the user to signal *yes* or *no*.

Use `yesno` in the following condition:

```
for (int which=0;yesno("input another grade?");which++)
```

If you use this before each iteration, the dialog box will come up and the user can decide whether to input another grade. Meanwhile, the user clicks Yes—the loop will go on and request more grades.

In this case, it is not strictly necessary to use the count variable `which`, since we don't need to count the iterations to stop the loop. However, this variable keeps track of how many grades were input and lets us know the number of students in class. Notice that using a different statement:

```
for(int which=1;which<class_size;which++)
```

would cause the value of the counter `which` to be one more than the number of grades when the loop finishes. It would then be necessary to decrement it before you use it to compute the average!

This solution does add inconvenience, because the user has to answer a *yes* or *no* question before each grade is input.

Finally, it is also possible to use a distinctive value to signal the end of the data. In this case, you should use an invalid value for the data to signal that there are no more data available. Since grades cannot be negative in this example, you could use a negative value to explain that all data has been input.

However, this solution will be correctly implemented only after you learn how to break out of the loop. SKILL ten presents these techniques.

Initial Conditions

Once the loop has been essentially designed, check whether all the initial conditions are correct. The total has to be initialized to zero.

Here is a complete implementation:

```
#include "franca.h"
void mainprog()           // c3avgrd.cpp
{
 float grade,total=0;
 float average;
 for (int which=0;yesno("input another grade?"); which++)
 {
   grade=ask("Enter next grade:");
   total=total+grade;
 }
 average=total/which;
 Cout<<average;
}
```

Rules and Regulations

This section summarizes rules for topics covered in the previous skills.

Compound Statements

A group of statements that starts with { and ends with } is considered a compound statement. For example:

```
for (int times=1;times<=10;times++)
// The lines below form a compound statement
  {
    Kim.JumpJack();
    Kim.say(times);
  }
```

You might have noticed that a function body is also a compound statement. For example:

```
void mainprog() // The lines below form a compound statement
{
  athlete Kim;
   Kim.up();
   Kim.ready();
}
```

Nesting

A compound statement may contain another compound statement. For example:

```
void mainprog()
{
  athlete Kim;
  for (int times=1;times<=10;times++)
  {
    Kim.JumpJack();
    Kim.say(times);
  }
}
```

The closing brace denotes the end of the compound statement. For this reason, no semicolon is needed.

Although the syntax does not require it, you should always indent inside a compound statement.

Scope

Any variables or objects declared inside the compound statement will be created inside the compound (when the point of declaration is first reached). They will be discarded when the compound statement is terminated. For example:

```
int i,j;
for (i=0;i<3;i++)
{
   int m;
   m=i+3;
}
```

In the sequence above, `m` is declared inside `for`. This variable will be created when the loop starts, and it will remain valid until the end of the loop (if the loop is restarted later, the variable will be created again). It is not possible to know the value of `m` after the loop has ended. Any references to `m` will cause a syntax error.

Even though the declaration of `m` is inside the loop, the variable is not created again for each iteration. It is created only once.

Simple Conditions

Conditions are expressed by means of relational expressions. As the name implies, the expression explores the relation between two variables or objects. A relational expression causes some kind of comparison to be made between two objects, and generates a result.

The result, which we expect to be either true or false, is actually represented in C++ by an integer. A value of zero denotes *false*. Any other value, usually *one*, denotes a true result.

The condition is always enclosed in parentheses.

Equality vs. Assignment

Special attention is needed when you compare to determine equality. When you check whether `a` is equal to `b`, it is a common mistake to write the following:

```
if (a = b) ... ;
```

In C++, this expression is acceptable in terms of syntax, but it does not compare `a` with `b`. In fact, this is an assignment expression, and the value of `b` will be brought into `a` (the original contents of `a` will be lost). The correct comparison should use the following equality operator:

```
if (a==b) ... ;
```

The expression `if(a=b) ...` will be accepted by the compiler and will cause a bug if you use it unintentionally. If the value in `b` is nonzero, the result will be equivalent to *true*. If the value in `b` is zero, the result will be equivalent to *false*. Even if you are experienced, you should avoid this type of expression.

Mixing Arithmetic and Relational Operations

Although you are advised against so doing, it is possible to mix arithmetic and relational expressions. For example:

```
k = 1+ a < b;
```

is correct in terms of syntax. If `a` is indeed less than `b`, the result of the logical expression will be 1, and the value of `k` will become 2. On the other hand, if `a` is not less than `b`, the value of `k` will become 1. If you mix expressions, a program may result that is hard to understand.

while loops

The format of the `while` statement is as follows:

```
while (  relational expression )  statement
```

in which the statement may be compound. The resulting action is that the relational expression is evaluated and, if the result is true, the statement will be executed. The evaluation of the expression and the execution of the statement will be repeated until the result of the expression is false, in which case the next statement will be executed.

If the statement is not compound, it is not necessary to enclose it in braces, and it may be written on the same line as `while`.

I strongly encourage you to continue to use braces. A common mistake occurs when you want to include another statement in the loop and forget to include the braces. Also, notice that there is no semicolon after the expression.

do/while loops

The format of `do/while` is as follows:

```
do statement while (relational expression);
```

The statement is executed, and then the expression is evaluated. If the result is true, the statement will be repeated, as will the evaluation of the expression. The repetition will end when the expression is evaluated as false.

for loops

The `for` statement provides complete control over a repetition and allows initialization, testing, and counting. The general form is as follows:

`for(`*`initial expression ; condition ; final expression`* `)` *`statement`*

in which the following is true:

- The initial expression is executed only once before the loop begins. It is useful for initializing variables.
- The condition is a conditional expression that will be evaluated once before each loop iteration. If the expression is true, the loop will be executed.
- The final expression is executed at the end of each loop iteration. It is useful to increment variables and to keep count of the iterations.
- The statement is any statement (including a compound statement). This statement is what will be executed at a loop iteration.

Absent Expressions

Expressions can be omitted inside the parentheses. It may be easy to see what happens if the initial expression and/or the final expression are missing. However, if you omit the loop condition, it is the same as having always a true condition! The piece of program below is an infinite loop:

```
for (;;)
Sal.say("I am stuck!");
```

Comma Expressions

Whenever an expression can be used, C++ allows you to use more than one expression, separated by commas. For example:

```
for(int done=0,int to_do=10; done<=howmany; done++,to_do--)
```

will cause the variables `done` and `to_do` to be created and initialized to 0 and 10 respectively, before the execution of the loop. At the end of each loop iteration, `done` will be incremented and `to_do` will be decremented.

If a value that resulted from the evaluation of the expression is to be used, the value of the leftmost expression will be used. For example:

```
howmany= 3,1,2;
```

will assign a value of 3 to the variable `howmany`.

Avoid the use of multiple expressions separated by commas. This will result in programs that are hard to understand.

Variable Declarations

It is possible to declare and initialize a variable in the `for` statement. This may be useful when you need a variable only during the execution of the loop. For example:

```
for (int done=1; done<=howmany; done++)
```

could be used instead of the following:

```
int done;
for (done=1;done<=howmany;done++)
```

Variables declared in the initial expression of `for` are considered to be declared before the compound statement that defines the loop. Therefore, their scopes remain valid after the loop has been terminated.

Try This for Fun…

- In the piece of program below, what is the value displayed on the screen? Why?

```
#include "franca.h"
void mainprog()
{
 athlete Sal;
 int done=5;
 {
  int done;
  for (int done=1; done<=10; done++)
  {

   Sal.ready();
   Sal.up();
  }
 }
 Sal.say(done);
}
```

Are You Experienced?

Now you can…

Identify repetitions that change pattern

Design loops in your programs

SKILL

NINE

DEVELOPING BASIC APPLICATIONS

- Implementing a point-of-sale terminal
- Exploring the design process

You are now going to work on your skill of developing applications by developing a simple point-of-sale terminal (such as what you see in stores and supermarkets). As you improve your skills, more complexity will be added to this terminal, and other applications will be used.

This short project will wrap up some of the skills you have developed so far, and will also discuss some of the issues involved in solving a real-life problem, in which many decisions are left to the designer.

Building a Point-of-Sale Terminal

Suppose that you are hired to develop software to implement a point-of-sale terminal. This software should enable a computer to keep track of sales, compute change, and handle other related issues.

Although your programming abilities are still limited, you can already solve these problems.

Often, in real life the client (the store owner) does not provide a good specification of the product he or she wants. The client knows it is too much trouble to do all this work manually and suspects that a computer can help. It is up to the designer (this means you!) to come up with a reasonable solution.

For most point-of-sale terminals, a bar code reader and a small printer are standard equipment. Regrettably, all we have is a standard computer—not even a printer! Your client (the clerk) will have to input the purchase price for each item by typing it from the keyboard.

The software you are about to create should add the prices of all the items in a single sale and write the total to the screen (after including the sales tax). If each customer made only one purchase at a time, you could solve the problem very easily. In fact, you could simply use one of the programs already seen in SKILL seven—`c3store1.cpp`.

However, you will now have to input and add several prices while handling the same customer. Only when all the prices for that sale have been input can you finish the sale by computing the tax, totaling the prices, etc. Besides, your software should enable the computer to handle one customer after another, instead of only one.

Designing Software for Multiple Transactions

Once you understand the basic operation, you may try to write a general algorithmic description of the problem. Here we go!

Clearly, the sales procedure will be repeated for each customer; therefore, you may consider the problem as follows:

Repeat while there are customers:

Input and add item prices.

Compute and display total with tax.

Compute and display change.

This looks like a correct, although still imprecise, description of what you want to do. It is fine as a start, though.

You may notice that inputting and adding item prices will, of course, involve another repetition. You may decide to explain right now how this is done, or you may decide it is a good idea to design a function to read all the items in a sale and to provide you with the total. This will simplify the main program and eliminate the need for an inner loop.

For the time being, we may think of a function that returns a floating point value to contain the total purchase (before tax) for a given customer. It is a good idea to establish a header for this function. For example:

```
float getitems();
```

Another question you may have at this time is, How should I display these values?

You could use the standard `Cout` as a solution. However, everything you write with `Cout` is erased with the next output. This may prove inconvenient for the store clerk. Another approach is to design a few boxes to display specific information: total price, amount tendered, change, etc.

In fact, we will design how your point-of-sale terminal uses these boxes right now.

Display Boxes

Here are some suggestions for possible display boxes:

- Current total—displays the total price of items entered so far (before tax)
- Sales total—displays the total sales price the customer is supposed to pay (including tax)
- Amount tendered—displays the amount tendered by the customer
- Change—displays the change to be given to the customer

These boxes can be declared as follows:

```
Box Cur_total("Current total");
Box total("Total Sale:"),Amount("Tendered:");
Box Change("Change due:");
```

The declarations above can be placed at the beginning of the program.

Now, if you look back to the general description, you will have no problem computing the total sales price with tax, inputting the amount, and displaying the change. The general description can be expanded either in algorithmic form or in program form. The complete loop body for each customer could be the following:

```
// Input and add items for each sale.
// Then include tax:
saletotal=getitems()*salestax;
// Display the sales total:
total.say(saletotal);
// Ask for amount tendered and display it:
amount=ask("Enter amount tendered:");
Amount.say(amount);
// Compute change and display it:
change=amount-saletotal;
Change.say(change);
```

The piece of program above explains what we want to do with each customer. It may be enclosed in a loop. The next question is, How do we determine how many times this loop is repeated?

Remember there are several ways to determine when the loop ended. I hope you realize is it not possible to determine the number of customers in advance. In our case, the most appropriate solution is to ask whether there are more customers. This is very easy:

```
while(yesno("Is there another customer?"))
```

If you think it is useful to keep track of the number of customers, you can increment a variable `customer` in a `for` loop:

```
for(;yesno("Is there another customer?");customer++)
```

Initial Conditions

To complete the loop, you should determine the initial conditions. The loop that takes care of each customer does not require a special initialization. However, the processing of each customer sale, as performed by the `getitems` function, does. Remember that this loop will be executed for each customer. Anything that was computed and/or left displayed on the screen for a previous customer must be cleared. This means that all the boxes should be initialized.

```
saletotal=0;
amount=0;
change=0;
Cur_total=0;
Saletotal.say(saletotal);
Amount.say(amount);
Change.say(change);
Cur_total.say(0.);
```

These statements not only reset the values to zero, but also display these values in the appropriate boxes. You can also clear the boxes instead of writing zeros. For example:

```
Amount.say("");
```

The *getitems()* Function

Essentially, the `getitems()` function is supposed to keep asking for item prices that belong to the same customer. These item prices should be added up and returned as a result. You can use the same approach to control the loop.

Here is a possible implementation:

```
float getitems()
{
  float total=0;
  for(int item=1;yesno("another item?");item++)
     {
       total=total+ask("Enter item price:");
     }
  return total;
}
```

Don't you think it would be nice if you could display the current total as the item prices are entered? This would look just like a real point-of-sale terminal!

However, this implementation poses a problem. If a box is declared to display this output, a new box will be created each time this function is called. Even though these boxes will be discarded at the end of each function execution, each new box will be assigned to a different location on the screen.

Use a box parameter.

To solve this problem, you need to use a box that is declared outside the scope of the function. You may declare the box in `mainprog` and pass it as an argument, preferably by reference. Remember that if a box is passed by reference, the function will not create a new (copied) box in a new location. The original box that was declared in the program will be used instead. You may experiment with passing the box by value to see what happens.

The updated code for the function would then be as follows:

```
float getitems(Box &display)
{
  float total=0;
  for(int item=1;yesno("another item?");item++)
     {
       total=total+ask("Enter item price:");
       display.say(total);
     }
  return total;
}
```

The function call must also be updated in `mainprog`:

```
saletotal=getitems(Cur_total)*(1+tax);
```

Finally, you may be wondering how to set up the sales tax information. It is clear that you could define this value in the program as a constant, but then every time the tax percentage changes, you would have to update the program. This is not smart. You may consider asking for the value of the tax percentage at the beginning of the program, and then keep that value throughout the day. This can be easily accomplished.

```
tax=ask("Enter the sale tax %")/100.;
```

The Complete Program Listing

Here is a complete listing of the updated program `c3sale2.cpp` (see also Figure 9.1):

```
#include "franca.h"                    // c3sale2.cpp
// Program to implement a point-of-sale terminal:
//      Gets tax information for each customer.
//      Reads item prices.
//      Totals each sale.
//      Computes change.
float getitems(Box &display)
{
  float total=0;
  for(int item=1;yesno("another item?");item++)
     {
       total=total+ask("Enter item price:");
       display.say(total);
     }
  return total;
}

void mainprog()
{
  Box Cur_total("Current total");
  Box total("Total Sale:"),Amount("Tendered:");
  Box Change("Change due:");
  float tax,amount,saletotal,change;
  dailytotal=0;
  tax=ask("Enter the sale tax %")/100.;

  for (int customer=1;
       yesno("Is there another customer?");customer++)
  {
     // Start one sale:
     saletotal=0;
     amount=0;
     change=0;
     Saletotal.say(saletotal);
     Amount.say(amount);
     Change.say(change);
     saletotal=getitems(Cur_total)*(1+tax);
     Saletotal.say(saletotal);
     amount=ask("Enter amount tendered:");
     Amount.say(amount);
     change=amount-saletotal;
     Change.say(change);
  }
}
```

percen the 8.25

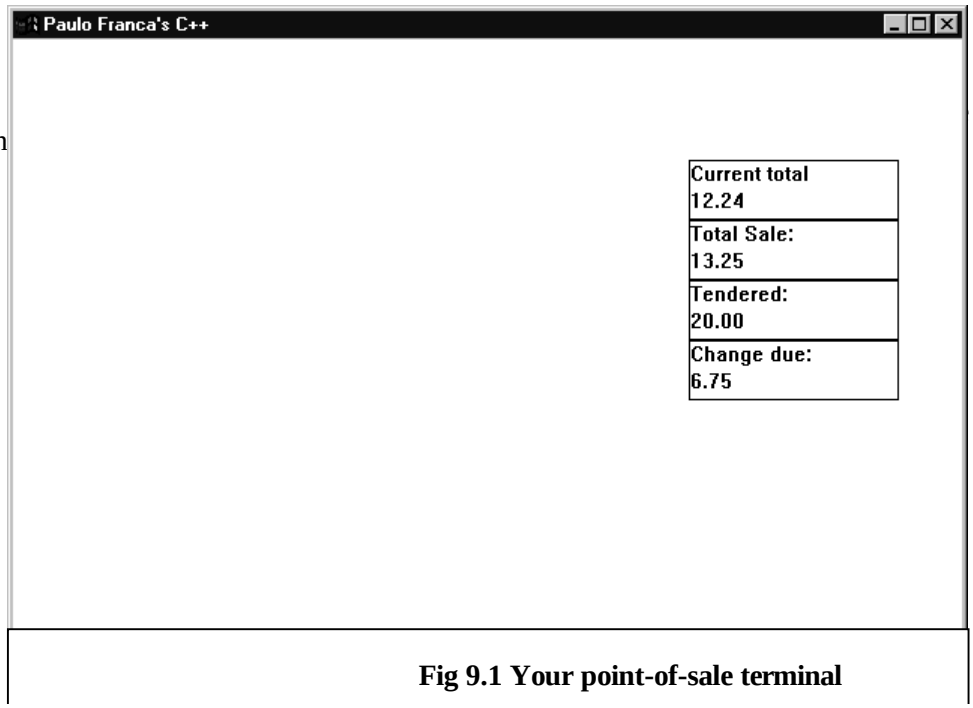

Fig 9.1 Your point-of-sale terminal

Exploring the Design Process

The design process is often less structured than you may at first be lead to believe. I could have presented the design steps in a more appropriate sequence. However, the truth is that this is not what will happen when you are creating your designs. For example, when you initially think of using a function (such as `getitems`), you may not have a complete idea of what you need. In this case, I tried to show you that it is only when you write the function code that you notice a parameter is needed. This may happen often in real life. No problem—just go back and revise what is needed.

The more you understand the problem and the available tools, the better you can solve it.

The User Interface

Do not neglect the user interface. The *user interface* is the dialog your program uses to communicate with the user. This dialog may include visual aids, sounds, and text. Keep in mind your average user. Make your interface easy and attractive. This will reduce the risk of input errors, and will keep your user happy with your service.

On the other hand, avoid asking too many questions or offering unnecessary explanations during program execution. This may be extremely annoying to your user.

Improvements and Modifications

After you have solved what you initially proposed, you may still notice several things that you can do to deliver a more useful solution to your user. For example, you could detail the total number of items per sale. You could offer a daily total and the number of customers after the last sale.

Last, but not least, the user will request modifications. No matter how perfect your program is, changes will be needed sooner or later. If you can anticipate some of these changes, go ahead and make them. As you prepare your program, keep in mind that you may have to change it. Make it easy to understand and modify.

Try This for Fun…

- Modify the point-of-sale program to do the following:
- Display the number of items in each sale
- Display the total number of customers and the total value of sales after the final sale

Are You Experienced?

Now you can…

Design a simple application

Design a suitable user interface for your applications

Understand some of the design trade-offs