PART II

GETTING SMART

In Part II, we will develop the skills of using functions and simple arithmetic expressions. You will learn how to use functions and arguments to make your computer smarter and how to manipulate values in functions to return a result. Also, you will further develop your problem-solving skills by using functions.

At the end of Part II, you will find a formal discussion of the rules that regulate the use of functions and numeric expressions.

SKILL

FOUR

USING FUNCTIONS

- Creating functions
- Using arguments to add versatility to functions
- Creating header files
- Understanding scope and scope rules

With the help of Sal, our tireless athlete, you will learn how to build simple functions. Any fitness exercise you may want Sal to perform can become a function. For example, once you teach Sal how to do a jumping jack, you will not have to teach him it again. All you will have to do is create a function that explains how to do a jumping jack.

Next, you will use arguments to increase the usefulness of functions. You will also learn how to keep useful functions stored in header files, so that you can use them with any of your programs.

Finally, an important concept will be covered: scope. Because each function is treated by the compiler as a separate entity, the names you choose for your variables and objects are valid in that function's scope. This is a nice feature in that it relieves you of having to keep track of all the names used in the program. However, scope rules must be clearly understood to avoid confusion.

What Is a Function?

Sometimes a group of orders constitute a distinctive action. You may want to simply refer to that group, instead of having to list, one by one, all the instructions needed to perform that action.

For example, take the following group of orders:

```
Sal.up();
Sal.left();
Sal.up();
Sal.right();
Sal.up();
Sal.ready();
```

We can see that this group constitutes one particular fitness exercise. If we let the computer recognize this sequence by a name—for example, leftright—whenever we say *leftright*, we actually want Sal to perform the sequence specified above. Generally speaking, we define a sequence of statements like this one as a function that explains how to perform that sequence. A similar feature in other languages may be called a procedure or a subroutine. Although C++'s nomenclature gives these pieces of programs the name *function*, the generic term *procedure* may still be used.

A *function* is a group of statements to which you give a name. These statements can be performed by calling the function name.

When you use functions, you make the computer smarter. Once you have explained the function, you don't need to repeat the orders: Your computer now knows how to carry out these orders! The program will look more understandable, too.

Unlike humans, computers will not remember the function forever. You must include the function code in each program in which you intend to use it.

Did you notice that the sequence of movements Sal performed constitutes one particular exercise? Sometimes fitness instructors will give a name to an exercise such as the one that consists of the following movements:

ready
up
ready

Fitness instructors have named this sequence of actions *jumping jack*. So, when the instructor wants the class to perform a jumping jack, he or she doesn't have to re-explain the steps.

For example, once the instructor has explained how to do a leftright and a jumping jack, they may give orders as follows:

jumping jack
leftright
jumping jack

Notice how much simpler the directions have become! Instead of giving a one-by-one description of all the steps involved, the instructor simply names what exercises they want performed.

This strategy applies to computer programming, as well: You may group several instructions and give them a name. When you want that function to be performed, you, the almighty programmer, merely need to mention the function name and the computer will perform each order described in the function.

Functions can be used with the robot, as well. For example, you can teach the robot how to draw a line of a given size.

Creating a Function

We can create a function right away. It is always a good idea to know exactly what you want to do before you start writing any program. So, let's get started by writing a description of what we want to do in plain English:

This is the function for a jumping jack:

Make Sal go up.

Make Sal return to the ready position.

That's all, folks!

In C++ language, this description would become the following:

```
void JumpJack()
{
     Sal.up();
     Sal.ready();
}
```

Naming the Function

We want to tell the computer that we are explaining how to perform a function, and we want to give the computer the name of that function. In the plain-English text, we used, *This is the function for a jumping jack*. This sentence explains that the function for a jumping jack consists of the sequence of instructions that follows.

Delimiting the Function

Which instruction is the first and which is the last in the sequence that belongs to this function?

You must clearly indicate the answer to this question to the computer, because other instructions may not belong to the function. To specify the first and last instructions, we included opening and closing braces ({}) around the sequence in which we were interested. The function starts after the opening brace and ends with the closing brace. In addition to including opening and closing braces, we also indented the instructions inside the braces. Indenting is not useful to the computer, but it is very useful for us; it is a good programming practice.

In the program, the function name follows the word `void`. We will explain this convention later in this skill. The function name is followed by `()`. There is no semicolon on the line that names the function

Technically speaking, the whole function is regarded as a single compound statement (compound statements are explained later). Inserting a semicolon after the function header and before the opening brace leads the compiler to believe that your statement ends there (before the actual function body).

Let's now look at the program `c2jmpjck.cpp`:

```
//                                        c2jmpjck.cpp
// Programmed by: Paulo Franca
// Revised January 5, 1996.
// This program uses a function JumpJack that exercises
// Sal once.
#include "franca.h"     // Include textbook programs
athlete Sal;            // Create an athlete Sal
void JumpJack()         // Function JumpJack starts here
{
  Sal.up();             // Up
  Sal.ready();          // Ready
}                       // Function ends here
void mainprog()         // Main program starts here
{
  Sal.ready();          // Ready
  JumpJack();           // Do a jumping jack!
}                       // Main program ends here
```

This program includes the code for the function `JmpJack` and the code for the function `mainprog`. Notice that `mainprog` is the name of the function in which you explain what you really want the computer to do. This function is like any other, except that it has to be present in every program you create. The `mainprog` function may use other functions, in the same way that it used the `JmpJack` function in the above example.

You must include the function before you can use it in the program.

If we want to have a couple of additional jumping jacks performed, we can simply include two more statements. `mainprog` will look like the following:

```
void mainprog()         // Main program starts here
{
  Sal.ready();          // Ready
  JumpJack();           // Do a jumping jack!
  JumpJack();           // Another
  JumpJack();           // And another
}                       // Main program ends here
```

Remember that a function is, essentially, a collection of actions that you have named. By using a function, you make your programs easier to read, and you also avoid having to explain the whole sequence if you want to perform it again. You can also make different objects perform the actions specified by a function.

Managing Several Objects

Suppose we have more than one athlete, Sal and Sally. We can easily use more than one athlete by including the following declaration in the program:

```
athlete Sal,Sally;
```

This tells the computer that we now have two objects that are of type `athlete` and, therefore, we may show both of them:

```
Sal.ready();
Sally.ready();
```

Now we want Sal to perform a jumping jack. We already know how to do that. What if we want Sally to do the same? It's of no use to call this function `JmpJack`, because `JmpJack` asks Sal to do all the exercises. (Would you like to try it? Go ahead!) What can you do? You may think of two simple solutions:

- You can make another `JmpJack` function specifically for Sally.
- You can list all the actions to make Sally perform the jumping jacks.

Both of these solutions are disappointing. We have already explained how to do jumping jacks, and they are not any different if performed by Sal, Sally, or anybody else. (After all, fitness instructors don't have to explain the exercises to one student at a time, do they?)

Using Arguments to Add Versatility

It would be great to rewrite the function to explain how *any* athlete performs a jumping jack. It is like explaining *this is the way somebody performs a jumping jack* without worrying who the *somebody* is that will actually perform the jumping jack.

In a sense, this is similar to a theater play. The author writes the play to consist of characters. The author seldom knows who will actually play the characters. When the play goes on stage, each actor takes the role of a character. However, you realize that characters and actors are not permanently tied together. The same character may be played by different actors, and a given actor plays the role of several characters in his or her professional life.

Arguments and Parameters

When we write a function, we don't know beforehand which objects will be used as arguments. Therefore, we refer to make-believe objects called *parameters*. When the function is called, an actual object will be used in place of the parameter. The actual objects, which you use when you execute the function, are called *arguments*. All this happens just like it does in the theater play.

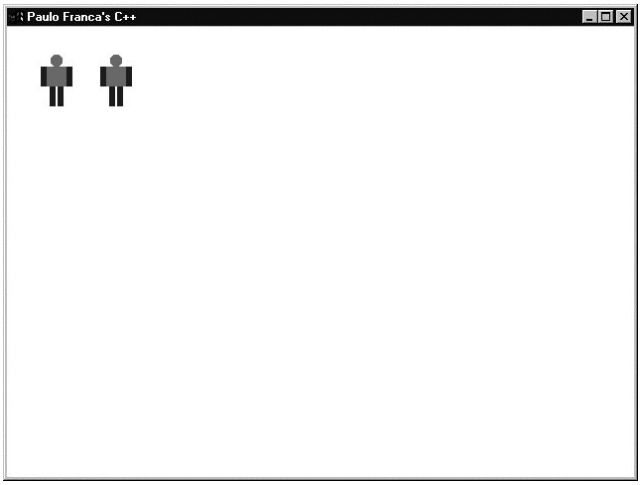


Fig 4.1 The result of c2jmpbdy.cpp

In the `JmpJack` function of the `c2jmpjck.cpp` program, we want to be able to give orders to Sal or Sally (or any other athlete that may show up later), instead of giving orders only to Sal. This task can be accomplished by making the function work with an impersonal athlete, instead of personalizing the task for one athlete:

1. Tell the computer how to make a fictitious athlete perform the jumping jacks. You may give this athlete a name, such as *somebody*. Include all the instructions to make *somebody* do a jumping jack. There is nothing called *somebody*—there is only Sal and Sally. *Somebody* is just a parameter.
2. Once you have explained how *somebody* can perform a jumping jack, tell Sally (or Sal) to play the part of *somebody* in that function. Sally is an object that really exists in your programs (like actors exist in real life) and can be used in place of the fictitious *somebody*.

Let's see how this works in `c2jmpbdy.cpp`:

```
//                              c2jmpbdy.cpp
//
#include "franca.h"
athlete Sal,Sally;
void JmpJack(athlete somebody)
{
     somebody.up();
     somebody.ready();
}
void mainprog()
{
     Sal.ready();
     JmpJack(Sal);
     Sally.ready();
     JmpJack(Sally);
}
```

Inside the function, we are using the object `somebody`, which, as we know, does not exist in reality. Nevertheless, we explain to this object what actions need to be done to have the jumping jack completed. If you look at the main part of the program, you will notice that the `JmpJack` function is used twice:

```
JmpJack(Sal);
JmpJack(Sally);
```

However, in the first instance, `Sal` is inside parentheses. What does this mean?

We want to execute `JmpJack`, but to actually use `Sal` as an argument in place of the fictitious parameter `somebody`. All the moves will be performed by Sal and not by the nonexistent *somebody*. The second time, we use `Sally` as an argument. Figure 4.1 illustrates the computer screen after this program is executed.

Give this new method for replacing arguments a try by declaring two athletes, Sal and Sally:

- Have them assume the ready position and say their names. Then, call the `JmpJack` function with each athlete as a parameter. Observe the action.
    - To do this, simply have Sal say “Sal” and have Sally say “Sally” before calling the `JmpJack` function.
- Now, call `JmpJack` first with Sally and then with Sal. Observe the result.
    - In other words, first call `JmpJack(Sally);` then call `JmpJack(Sal);`.

Default Values for Parameters

Sometimes it may be desirable to specify a default value for a parameter. This is especially useful when you want to use the same (default) object as an argument most of the time.

For example, suppose that in most cases we want to perform a jumping jack with Sal, and only a few times with Sally. We can specify a default value for the parameter in the function definition by appending an assignment to the parameter. For example, changing the function header `JmpJack` to the following:

```
void JmpJack (athlete somebody=Sal)
```

will cause `Sal` to be used as the default athlete if no other athlete is specified in the function call. Therefore, a statement such as the following:

```
JmpJack();
```

will cause the same result as if we had:

```
JmpJack(Sal);
```

This simply allows us to omit the actual argument when calling the function. It is still possible to include the argument when needed. For example, if we want `Sally` to be used, we can use the following function call:

```
JmpJack(Sally);
```

When specifying a default argument, the object specified as the default must be previously defined and known to the function (see the section on scope later in this skill).

Multiple Arguments

It is also possible to create functions that use more than one argument. For example, we can have another function that takes two athletes as arguments and have them both perform a jumping jack. In C++, it is possible to have two functions with the same name in the same program—provided that the functions have a different number of arguments or have different classes (or types) of arguments.

Let's develop a new `JmpJack` function that makes both athletes exercise. Of course, you may have thought of calling the existing `JmpJack` function for each of the athletes:

```
void JmpJack(athlete first, athlete second)
{
  JmpJack(first);
  JmpJack(second);
}
```

However, this will make the athletes exercise one at a time. Try it, if you'd like.

What can we do to make them both exercise simultaneously? Actually, it is impossible to perform two actions at the same time in this case, because the computer operates sequentially on your program. We have to control the time that the athlete remains in each position.

To give the illusion that both athletes are exercising at the same time, we can tell the first athlete to be ready for zero seconds, then the second athlete to be ready for 1 second, as usual. Then, we do the same with the other movements. Most computers nowadays will perform the movement so fast that you will hardly notice it. Only the picture that is kept for a long period (1 second) will actually be registered in your mind.

This is the algorithm:

Function JmpJack to make two athletes perform a jumping jack at the same time:

First athlete stays in *ready* for zero seconds.
Second athlete stays in *ready* for 1 second.
First athlete stays in *up* for zero seconds.
Second athlete stays in up for 1 second.
First athlete stays in *ready* for zero seconds.
Second athlete stays in *ready* for 1 second.

Can you finish the function and see the results? You may start with the following:

```
void JmpJack(athlete first, athlete second)
```

Originals and Copies: Value vs. Reference

In general, when you use a function that has an argument, the function will use a copy of the argument. In other words, when you issue a statement such as the following:

```
Jmpjack(Sally);
```

the computer makes a copy of the athlete `Sally` and then uses it in the `JmpJack` function. Your original athlete remains unaltered no matter what you do in the function. C++ usually does this when calling functions. You can be sure that the function does not tamper with your original object.

In the special case of an athlete, differences are small. Another athlete will be created in the exact position as your original, and this new athlete will move according to the function instructions. By looking at the screen, you cannot tell if the movements were performed by the original athlete or by its clone.

If you want to see the consequences of using a copy, you can use a `Clock` object. In the example that follows, we are going to use a `Clock` object called `timer`. We then invoke a function that resets the timer. Instead of resetting the original Clock, the function will reset the Clock's clone!

This can be easily observed by having a box "say" the time before, during, and after the function's execution. To do this, we can declare three boxes and label them `Before:`, `During:`, and `After:`. We can use each one to display the time at the appropriate moment. `c2clkcpy.cpp` illustrates the problem:

```
#include "franca.h"          //      c2clkcpy.cpp
// This program illustrates how arguments are copied
//   when used in a function.
Box before("Before:"),during("During:"),after("After:");
void zero(Clock clone)
{
   clone.reset();            // Reset the Clock
   during.say(clone.time());// Show the time (0.)
}
void mainprog()
{
   Clock timer;
   timer.wait(1.);           // Make sure time>1
   before.say(timer.time());// Show the time
   zero(timer);              // Call the function
   after.say(timer.time()); // Show the time
}
```

If you run this program, you should see three numbers on the screen. The first number is the number of seconds that elapsed since the program started. This should be close to 1 second, since `wait(1)` was included.

The second number is the number of seconds that elapsed since the Clock was reset inside the `zero` function. This number should be zero.

The third number is the number of seconds that elapsed since the Clock was last reset. You might expect this to be close to zero, because the Clock was sent to the function as an argument and was reset. But this is not true! The actual value you'll see is probably 1 or a little more than that. This indicates that your Clock was not reset: A clone of your Clock was reset instead. Check out Figure 4.2.

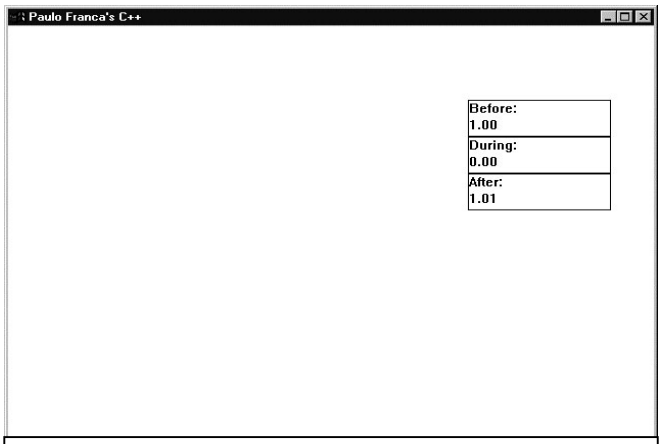


Fig 4.2 The result of c2clckcpy.cpp

This looks like a great idea to prevent you from messing up the contents of your objects when you invoke functions. But what if you want to operate on the original object?

Passing by Reference or Passing by Value

C++ allows you to indicate that your function is supposed to operate on the original argument and not on a copy of it. This is referred to as a parameter passing by *reference*, as

opposed to a parameter passing by *value*. Passing by reference is very simple. Indicate which parameters you want passed by reference (passing by value is the default). In C++, you simply precede the parameter name by the reference symbol `&`. In our example, the modified function would be as follows:

```
void zero(Clock & clone)
{
   clone.reset();           // Reset the Clock
   during.say(clone.time());// Show the time (0.)
}
```

This is the only change! The statements in the function remain the same, as do those in the main program. By simply including the reference symbol (&), you make sure that all operations will be performed on the original object. You can now modify the program and observe the results.

If you have more than one parameter, you must use a reference symbol for each parameter that you want passed by reference. Those that are not preceded by `&` will be passed by value.

When should we use references?

Clearly, you must use a reference when you want to use the original argument. There may be other situations in which using a reference may be convenient as well.

When a clone object is used, the computer spends some time preparing a copy of the object. In most cases, this time is negligible. However, some objects consume more time while being copied. `athlete` objects, for example, are relatively complex. If you're sure you don't need protection against modifications to your original and your object complexity is significant, you may also consider passing by reference.

Teaching an Object

You may have noticed that there is a similarity between invoking a function and sending a message to an object. In both cases, a sequence of actions is executed. When you send a message to an object, you invoke a function that is restricted to that class of objects. These are called member functions. Member functions will be studied in detail in SKILLs sixteen and seventeen.

When the `athlete` class was created, it was designed to respond to messages of `ready`, `up`, `left`, and `right`. This was done by including a member function for each of those actions. We might have included a function for a jumping jack, too, but we did not want to steal all the fun from you!

Member Functions and Non-Member Functions

Objects have functions that correspond to each message the objects are supposed to understand. For example, this means that `athlete` objects have functions such as the following:

ready
up
left
right
say

However, these functions are part of the objects and cannot be used independently. Formally speaking, these are *member functions* of the `athlete` class of objects. Member functions can be defined only when the class is being defined. We will learn how to do this in a later skill. On the other hand, the `JmpJack` function does not belong to any class and, for that

reason, we were able to create it at will. Functions such as `JmpJack` that are not part of a class are called *non-member functions*.

Header Files

If you have a function that you want to use in several programs, you may have to copy it to each new program in the place where you want to use it. This is not so smart! It would be good if you could tell the computer to grab the function that you want and copy it for you.

You may have stored the function in a file (just like any other program). You then use the `#include` directive to make a copy for you. This is the purpose of `#include`. I prepared several functions to make your life easier, but if you have to copy these programs one by one, you will soon give up. Instead, I stored these functions in a file (`franca.h`), and when you use a statement such as the following:

```
#include "franca.h"
```

you are telling the computer to find the file `franca.h` and copy it into your program. You can do this with your programs and functions, too!

If a function `jumpjack.cpp` is stored in a disk file, you can copy the function by including the following line in the program in which you want it to be copied:

```
#include "jumpjack.cpp"
```

As you Look at the program, you will not see the function `jumpjack` (just like you don't see the functions of `franca.h`). You will only see `#include`. Nevertheless, just before the compiler starts translating your program into machine language, a copy of the requested program will be placed where `#include` was.

For the compiler to find your included file, this file should be in the same directory as the project with which you are currently working (for example, `c:\franca`). If this is not the case, you may also specify the full path to your header file:

```
#include "c:\franca\jumpjack.cpp"
```

When you want to include files that are located in your directories, the file name is enclosed in double quotes, as above. In other situations, you may want to include files that are located in the compiler directories (as you will learn how to do in later skills). Then, instead of enclosing the file name in double quotes, you enclose it in angled brackets. For example:

```
#include <math.h>;
#include <iostream.h>;
```

What Is a Directive?

A *directive* is an instruction to the compiler. It specifies what you want done before your program is compiled. Directives are not part of the C++ language; they are mere directions that you provide to the compiler.

The compiler recognizes a directive by the pound sign (#) that is present as the first character in a line. There are several compiler directives, but we will only study `include`.

Creating Header Files

Files that contain pieces of programs to be copied to other programs are usually called *header files* (or *include files*) and are identified by the extension `.h` (instead of `.cpp`). However, you could include files with the `.cpp` extension, but to do the professional thing, you should store the file with the extension `.h`. You can do this by using File ➢ Store As or File ÿ Save As in your C++ compiler. Type the name of the file with `.h` right after it (refer to additional material on this topic at the end of this skill).

Try this exercise now: Transform the `JmpJack` function into a header file. Write a program that includes `jumpjack.h` and uses this function.

Understanding Scope

If you hear me saying, "I love Napoleon," chances are that you will understand I am a great admirer of the French general. However, if you are thinking about pastries, you may think instead that I am referring to the delicious pastry that bears the same name as the general.

This is what happens when we do not define our *scope*. You don't have to specify all the time whether you are talking about the general or the pastry, provided you are sure that you are in the right context—the same scope—as dictated by the conversation.

Scope is used in C++ and other programming languages to allow you to designate different objects using the same name, provided the objects are in different scopes.

If an object is defined in a function, it is not known outside of that function: it is *local* to that function. If the object is defined outside all functions, it is known by all functions that are defined after you have defined the object. It is then called a *global* object.

The `c2jmpjck.cpp` program contains a global object, `Sal`. The function `JmpJack()` simply used this global object to perform the exercise. This was a serious limitation because only one object could be used with the function.

The later version, `c2jpmbdy.cpp`, offers two global objects, `Sal` and `Sally`. The function `jmpjck` uses the parameter `somebody` and, therefore, is able to exercise any athlete that is passed as an argument. `Sal` and `Sally` are still left as global objects in this program, but they don't have to be! `Sal` and `Sally` can be declared inside the `mainprog()` function to be local objects.

Although it is not wrong to use global objects, it is a good idea to avoid their use. A better implementation of `c2jmpbdy.cpp` is shown below as `c2jmbd1.cpp`.

```
//                              c2jmbd1.cpp
//
// This program uses a function JmpJack that exercises
//   a given athlete (parameter).
#include "franca.h"
void JmpJack(athlete somebody)
{
     somebody.up();
     somebody.ready();
}
void mainprog()
{
     athlete Sal,Sally;
     Sal.ready();
     JmpJack(Sal);
     Sally.ready();
     JmpJack(Sally);
}
```

Notice that, in this case, `Sal` and `Sally` are known only inside the `mainprog()` function. This is perfect, since no other function needs to know them.

Repeating Local Object Names

A slightly different implementation of this program can be seen below. The only difference between this program and `c2jpbd1.cpp` is the parameter name used in the `JmpJack(athlete Sal)` function. Instead of using `somebody` to designate the parameter, this program uses `Sal`.

The following example has a scope of `Sal`:

```
void JmpJack(athlete Sal)
{
 Sal.up();
 Sal.ready();
}
```

This next example has a scope of `Sal` and `Sally`:

```
void mainprog()
{
 athlete Sal,Sally;
 Sal.ready();
 JmpJack(Sal);
 Sally.ready();
 JmpJack(Sally);
}
```

You should avoid using the same name to designate different things. However, the computer will not be confused if you forget. Inside the function, `Sal` will designate the parameter that will be substituted with a copy of the actual argument. This particular `Sal` plays the role of the other object `Sal`, which was declared in the `mainprog` function, and also plays the role of the `Sally` object.

This is the main benefit of scope: You don't have to keep track of all the names used in each function. Each name is only valid within the scope in which it was declared.

Another implementation (which should be avoided) is shown below. In this case, a global and a local object both share the name `Sal`. The program will still operate correctly, but inside the function `JmpJack` both objects have a valid scope. The locally declared object takes precedence over the global one. However, you may easily get confused when developing a program in which several objects have valid scope.

The following example has a scope of `Sal`:

```
athlete Sal;
```

This next example has a scope of another `Sal` and the original `Sal`:

```
void JmpJack(athlete Sal)
{
 Sal.up();
 Sal.ready();
}
```

The final example has a scope of `Sally` and `Sal`:

```
void mainprog()
{
 athlete Sally;
 Sal.ready();
 JmpJack(Sal);
 Sally.ready();
 JmpJack(Sally);
 }
```

When two objects or variables that use the same name are valid in a given scope, the one with the most local scope is used. No two objects with the same name can be declared in the same scope.

A common mistake for beginners is to forget that the parameter is already declared in the parameter list. For example, in the function below, there is an athlete `somebody` declared in the parameter list. The second declaration for an athlete with the same name is incorrect.

```
void Jump(athlete somebody)
{
   athlete somebody; // This is wrong! Somebody exists!
   ...
```

Examples of Scope

The following example shows an invalid declaration of two objects with the same name. The first declaration of `Sal` occurs inside the `mainprog` function. The function scope is delimited by the closing brace at the end of this listing. Therefore, it is not possible to declare another object `Sal` before the delimiting closing brace.

```
void mainprog()
{
  athlete Sal;
  jmpjack(Sal);
  athlete Sal;     // This is incorrect:
  ...              //   first declaration of Sal is
}                  //   in the same scope!
```

As we will see later, other C++ constructs are also delimited by braces. In this case, a new scope is defined for each set of matching opening/closing braces. The example below is correct. The second declaration for `Sal` is within a new scope delimited by a new pair of braces. This new object will exist only within this inner scope. References to `Sal` within this inner scope use this new object instead of the first one. Any reference to `Sal` that occurs before or after the inner braces uses the first object.

```
void mainprog()         // c2scope.cpp
{
  athlete Sal;
  Sal.ready();          // This will use the first
  {
      athlete Sal;      // This is correct: new scope!
      athlete Sally;
      Sal.up();         // Another athlete is used
      Sally.left();
  }                     // End of scope here!
  Sal.right();          // This uses the first again
}
```

In the example above, another object, `Sally`, was also declared in the inner scope. References to this object are valid only within that scope (remember that a scope is delimited by a pair of braces). Any reference to `Sally`, either before the inner opening brace or after the inner closing brace, is an error.

Are You Experienced?

Now you can…

Write functions to simplify your programming tasks

Use arguments to make functions reusable

Choose between reference or value when passing arguments

Create header files to use a function in different programs

SKILL

FIVE

USING NUMBERS

- Storing integers, floating point numbers, and alphanumeric characters in variables
- Manipulating numeric variables
- Using arithmetic and relational operators in simple expressions
- Inputting and outputting values

In several situations, we have to use numbers in our programs. Numbers keep track of how many times we are repeating something, note the time that has elapsed, and represent general quantities that we use in our programs.

As briefly mentioned in Part I, there are simple objects called variables. All they do is accommodate numbers.

The name *variable* originated from the fact that the value stored in a variable may vary during program execution.

Understanding Numbers and Numeric Variables

Variables are identified in a program by an identifier. Variable identifiers obey the same rules as object identifiers. A *variable* is a location (or set of locations) in the computer memory in which we can store a number. Numbers can represent characters, colors, or just about anything we want. C++ recognizes three fundamental types of variables:

- Integer: this type represents numbers that do not include a decimal part. There are two types of integers, `int` and `long`.
  - The type `int` can hold an integer number ranging from –32,768 to +32,767.
  - The type `long` can hold an integer number ranging from –2,147,483,648 to +2,147,483,647.
- Floating point: this type represents numbers that include a decimal part. There are two types of floating point numbers, `float` and `double`.
  - The type `float` can hold a positive or negative floating point number ranging from $3.4 \lozenge 10^{-38}$ to $3.4 \lozenge 10^{38}$. The number of significant digits is limited.
  - The type `double` can hold a positive or negative floating point number ranging from $1.7 \lozenge 10^{-308}$ to $1.7 \lozenge 10^{308}$. The number of significant digits is also limited.
- Character: this type, although it stores an integer number, associates that number with a code that represents an alphanumeric character. This is the type `char`. A variable of type `char` can also be used as an integer.

To use a variable in your program, you have to declare it. You declare variables just like you declare objects. Remember that you declare an object by giving the object class followed by the object identifier. For example:

```
athlete Sal;
Clock mytime;
```

You declare a variable by giving the variable type followed by the variable identifier. For example:

```
int  count, howmanytimes;
float elapsedtime;
```

In addition, you can specify an integer to be *unsigned* if you don't need negative numbers. In this case, you will have twice as many positive values. For example:

```
unsigned int lapcount;
```

would declare an object `lapcount` that could accommodate integers from 0 to 65,535.

You may be wondering why these limits are strange numbers, instead of neat thousands or millions. This has to do with the way that computers store numbers in their memory.

Why are there different types?

The main reason there are different types to represent numbers is that the different types make more efficient use of the computer's memory. Each type of variable is able to accommodate a given range of values. To represent a floating point number, we need four times the space needed to represent a character. Moreover, we cannot represent numbers that have an arbitrary number of digits. Each type has a specific limitation, as we shall see later.

All the numeric types obey similar syntax rules.

General Rules for Numeric Variables

Numeric variables obey the same rules that all objects obey:

- Use an identifier (a name) to designate the variable.
- Use these identifiers in the same way you have already used them for other objects.
- Declare the variable before you use it.
- As with any other object, declare a variable by preceding the variable name (identifier) with the type.

The declaration of a variable does not imply that it has been assigned a value. It only implies that you can use this object in your program. For example, when an `int` is declared, the computer simply saves a piece of memory to store the value, but does not set this value to zero or any other value.

Variables can be used in several ways:

- You can check the value of the variable.
- You can assign a new value to the variable.
- You can compare a variable to another.

Values

We will now experiment with the following code listing:

```
#include "franca.h"
void mainprog()
{                          // one, two, three program
   Athlete Sal;
   int one, two, three;
   Sal ready(5);
   Sal.say("one");
   Sal.say(one);
   Sal.say("two");
   Sal.say(two);
   Sal.say("three");
   Sal.say(three);
}
```

What are the values of `one`, `two`, and `three`? If you try to run this program, you will probably notice some strange numbers when `Sal` tries to tell you the values of the integer variables. This is because we have not set the variables to any specific values.

The computer's memory is always storing some information that was not cleared after use. If you use a variable without storing any value in it, the computer uses whatever information remains at that memory location.

Assigning a value

Use the assignment operator (=) to assign a value to a variable. This symbol looks like the equals sign you know from math. To assign a value, give the variable identifier, followed by = and then the value that you want.

For example, to make a variable `howmany` assume the value `21`, we could use the following:

```
howmany=21;
```

Of course, the variable `howmany` must be declared before it is used:

```
int howmany;
```

It is important you understand that in C++ the symbol = means *becomes* or *receives the value of*—it represents an action that takes the value on its right side and stores it in the variable whose identifier is on its left side. This is not the same as an equation in math! For example, the following is not valid in C++:

```
21=howmany; // This will never work!
```

because the compiler would try to fetch the value of `howmany` and store it in `21`. This does not make sense. 21 is not a variable and should always remain 21!

Here is another example: Suppose you have two integer variables in your program whose identifiers are `new` and `old`. Also, suppose `new` has a value of `15` and `old` has a value of `13`. If you execute a statement such as the following:

```
new=old;
```

both objects will now be equal to `13`. This is because the computer takes the value on the right side and stores it in the variable whose identifier is on the left side of the assignment operator. If, in addition, you have a statement such as the following:

```
old=new;
```

the result will be that both objects still have the value `13`, because `new` just had its value changed to `13` with the previous statement.

If you have two variables, `new` and `old`, and you want to exchange their values, the following sequence is incorrect:

```
new=old;
old=new
```

because when you copy the contents of `old` into `new`, the previous contents of `new` are lost. To correct this problem, you need an additional variable to keep the original value. For example:

```
temp=new;
new=old;
old=temp;
```

Initialization

It is also possible to assign a value at the time that it is declared. For example:

```
int maybe=21;
```

will not only create the integer variable `maybe`, but will also assign the value `21` to it.

In the `one, two, three` program, you can initialize the values by changing the declaration to:

```
int one=1,two=2,three=3;
```

You can go ahead and try this now.

Using Arithmetic Operators in Simple Expressions

Variables can also be used in expressions by using the arithmetic operators for addition (+) and subtraction (–) These operators have the meaning you would expect. Variables can also be multiplied (*) and divided (/), but we will deal with these later.

You can combine integer variables and integer numbers in an expression to produce a result. This result can be stored in any variable if you use the = operator.

For example:

```
int maybe,hisage,difference; // Declares int object
maybe=21;                    // Stores the value 21 in maybe
difference=5;                // Stores the value 5 in difference
hisage=maybe+difference;     // Adds the value in maybe to
                             //      the value in difference and
                             //      stores the result in hisage
```

will result in the value `26` being stored in `hisage`.

Incrementing and Decrementing

You can also increment (add one) to your variables. This is very easy—you can do something such as the following:

```
maybe=maybe+1;
```

This is correct, since the computer picks up the current value of `maybe`, adds one to it, and stores the result in the same place again. (Can you see how this is different from a mathematical equation?) Similarly, you can decrement any value by doing the following:

```
maybe=maybe-1;
```

which is also correct.

The ++ and – – Operators

In C++, there is another possibility: You can increment a variable by following the identifier with the increment operator `++`. Do the following:

```
maybe++;
```

There is one important difference with this increment operator: You can use it inside another expression.

```
maybe=25;
hisage=maybe++;
```

In this case, the value in `maybe` is used first, and only after this will `maybe` be incremented. In the example above, the values will be the following after execution:

- `maybe` will be `26` (because it was incremented)
- `hisage` will be `25` (because it was copied from `maybe` before `maybe` was incremented)

If we have the following sequence instead:

```
maybe=25;
hisage=++maybe;
```

`maybe` will first be incremented to become `26`. Then, this result will be assigned to `hisage`. This will cause both `maybe` and `hisage` to become `26`.

A third possibility is the following sequence:

```
maybe=25;
hisage=++maybe++;
```

What do you think the values of `maybe` and `hisage` will be? `maybe` is incremented twice—once before and once after it is used in the expression!

Similar results can be obtained with the decrement operator `--`.

`++` and `--` can be used *before* or *after* a variable. If they are used before, the variable is first incremented (or decremented) and then used in the expression.

As an exercise, consider that the following declaration is included in a program:

```
int i=1, j=2, k=3;
```

Use the expressions below to indicate the value of *k* after the expression is executed. If the expression is incorrect, specify the reason. Assume that these statements are executed in sequence.

```
k=++k+j;
k=i+j++;
k=i-j;
i+j=k;
k++=j;
```

Using Relational Operators

You can also compare values by using relational operators. In general, when you make a comparison, you want to know whether a value is any of the following:

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

Comparisons are useful to control the number of repetitions (as we will see in the next section) and also to make decisions in programs (as we will see in a future skill).

OBJECTS VS. VARIABLES

In many cases, we will refer to objects and variables in a similar manner. In fact, there is little difference between the two things. In the good old days of computing, there were only variables. Variables could hold information, and you could perform a limited number of operations using variables. Objects are a recent creation. Besides storing information in objects, you can specify the kinds of operations you perform with your objects. In other words, you can specify what kinds of messages to which your object will respond. Therefore, an object is a "smarter" kind of variable, because you can teach the object to do new things.

Inputting Values

Quite often, you may have to bring information into your computer. This information may be *input* through the keyboard by a human operator, may be read from a previously recorded disk file, or may be acquired by the computer from a sensor of some kind (for example, the clock). Information that comes from the real world outside the computer is valuable. Computers can respond to a variety of situations because of input.

Generally speaking, input occurs only when the program requests it. Input information is usually assigned to a variable or an object, so that the program can test it and use it.

In this skill, we will study only means for inputting information from the keyboard. Input from disk files will be examined in SKILL twenty-three. The input methods studied in this skill can be used only with the support software you downloaded from the Sybex Web site.

General techniques for inputting and outputting in C++ will be discussed in Part VIII.

It is possible to request that a value be input from the keyboard and then assigned to a variable. For example, when we want to make a purchase and receive some change, it is impractical to create a program each time we want to purchase something. Instead, we instruct the computer to request the value of the purchase and the amount paid.

We will examine two ways to input values:

- `ask` is an easy method that is available only with our software files.
- `Cin` is also available with our software files, but closely resembles the standard method used in C++.

The *ask* Input Function

The `ask` function generates a dialog box that asks the user to input a value from the keyboard. The value is then made available, so that you can assign it to a variable. The syntax is as follows:

```
ask ( question );
```

`question` represents the sentence that will be displayed to the user. For example:

```
Price=ask("Please enter the price:");
```

will cause a dialog box (such as the one shown in Figure 5.1) to be presented to the user. After the user types the value and clicks OK (or hits Enter), the dialog box disappears and the value is assigned to the variable—`Price` in this case.

The Cancel option is nonfunctional in this dialog box.

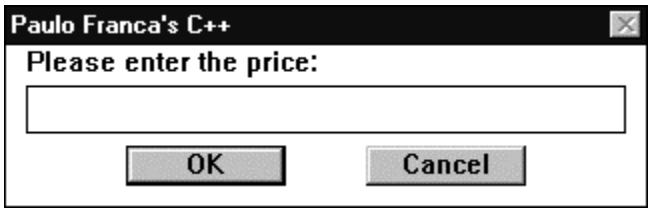


Figure 5.1: Asking for input

The *Cin* Simulated Input Stream

The other alternative you can use to input values is the object `Cin`, which simulates the standard method for C++ input. The syntax in this case is a bit different from the `ask` function:

```
Cin >> variable name ;
```

`variable name` is the identifier of the variable in which you want the value to be stored. The main inconvenience of this approach is that there is not a question presented to the user. However, you can display a message before inputting the value (see the following section on outputting values).

The effect of `Cin` is similar to that of the `ask` function. A dialog box that instructs the user to input either an integer, a floating point number, or a sequence of characters will also be displayed (see Figure 5.2).

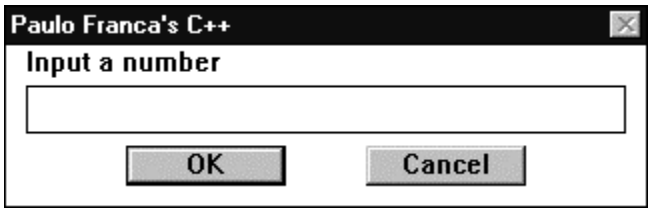


Figure 5.2: Inputting values with `Cin`

Outputting Values

In the same way that there are situations in which you want to input information to the computer from the outside world, there are other situations in which you need the computer to let you know something. Information that the computer provides to the outside world is called *output*.

Output can be received by writing numbers or characters to the screen or printer, recording information to a disk, displaying a picture on the screen, playing a sound, and many other ways. Some of these methods have been explored earlier in the book. In this section, we will examine only output that is written to the display screen as numbers or characters.

You already know how to display a value or a message by using athletes and boxes. Of course, the sentence that can be displayed is very limited in these cases. Although boxes are not generally available in standard C++ output, their functionality is similar to what you will find when you leave the standard text interface behind and move to real graphic-interface programming.

There is a `Cout` object to display program output that is similar to `Cin`. This, too, is implemented in our software as a simulation of the standard output used in C++.

The *Cout* Simulated Output Stream

The `Cout` output produces a wide rectangular box at the bottom of the screen and writes our output to that location. Each new output erases the previous one, since there is only one box to display.

The syntax of `Cout` is similar to that of `Cin`. However, the direction of the arrows is reversed. This is supposed to help you remember that the values go *from* the variable *to* `Cout` and *from* `Cin` *to* the variable. For example:

```
Cout<<  variable name ;
```

We can display the value of the variable `Price` by doing the following:

```
Cout<<Price;
```

The program below uses `ask` and `say` (see Figure 5.3):

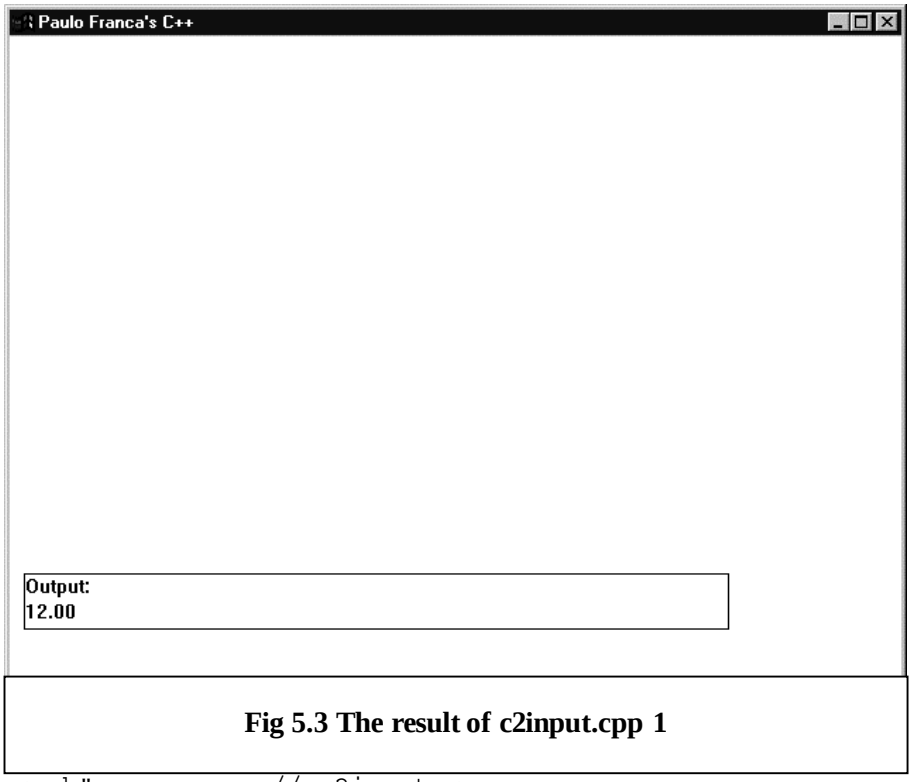


Fig 5.3 The result of c2input.cpp 1

```
#include "franca.h"           // c2input.cpp
athlete Julia;
void mainprog()
{
  float Price;
  Julia.ready();
  Price=ask("Please enter the price:");
  Julia.say(Price);
}
```

Standard C++ inputting and outputting is performed with `cin` and `cout` (lowercased). These will be studied in Part VIII. It is not possible to use `cin` and `cout` while using the Windows graphic interface in the same program. `Cin` and `Cout` (capitalized) are included in the support software to use as an alternative while using the Windows graphic interface.

The next example uses `Cin` and `Cout`. (It is possible to mix all these functions in the same program.)

```
#include "franca.h"           // c2cinout.cpp
void mainprog()
{
  float Price;
  Cout<<"Enter the price:";
  Cin>>Price;
  Cout<<Price;
}
```

In this program, since `Cin` makes no provision for a message, we used `Cout` to display a message to the user. It is always a good idea to tell the user what kind of input you are expecting. The final screen for this program is shown in Figure 5.3.

We can also use this program to illustrate the use of simple expressions. Let's suppose that first we want to compute the price after the sales tax is added. The program can be modified as follows:

```
#include "franca.h"
void mainprog()
{
  float Price,tax=8.25;          // New variable here
  Cout<<"Enter the price:";
  Cin>>Price;
  Price=Price+(Price*tax/100)    // Compute new price
  Cout<<Price;
}
```

There is a new variable, `tax`, initialized to `8.25`, which assumes a sales tax of 8.25 percent. After reading the `Price`, the program computes a new value for `Price` that includes the sales tax.

Are You Experienced?

Now you can…

Manipulate numeric variables

Input numeric values from the keyboard

Output numeric values to the computer screen

SKILL

SIX:

USING FUNCTIONS TO SOLVE PROBLEMS

- Returning values as results of a function
- Using inline functions
- Solving problems

Now that you have learned how to handle numeric variables, you can add to your skills in creating functions. In this skill, we will examine a few more techniques for writing and using functions, and we will reexamine the issue of problem solving. However, this time you have the powerful knowledge of using functions to simplify your problem-solving tasks.

In a real-world situation, it is unlikely that someone will tell you to "write a function to do this" or to "write a function to do that." You will be faced with a problem and you will have to do your best to solve it, using functions or not. In fact, most of the problems you will see in real life are not well defined at all!

A customer may not know exactly what he or she wants (sometimes they may not even know what the problem really is). You may be responsible for studying the problem and offering a solution that will help you and your customer.

Functions play an important role in problem solving. There are many benefits to using functions in your programs:

- Functions simplify the main body of the program.
- Functions make it easier to develop, understand, and modify the program.
- You may already have some functions available that can solve parts of the problem.

Always use what is available. Any new piece of program that you create will take time to develop and debug. If there is no function available to help you, start developing functions that can be used in the future to solve similar problems. You will be amazed by the amount of work you can save. Many problems in real life are variations of another problem.

It is up to you to choose how deeply into functions you want to go. In general, you should do the following:

- Deliver a program that solves the specified problem.
- Write as few lines of code as possible.
- Make your program reusable.
- Reuse whatever you can.
- Make your program easy to understand and modify.
- Deliver your program on time.
- Deliver your program as free of errors as possible.

In addition, it is important that you document all your programs and functions. Include a reasonable amount of comments with the code, and write separate documentation (like a user's manual), if necessary.

Returning Values in a Function

In general, every time you call a function, it causes some kind of result. The result may be an animation shown on the screen (such as in the jumping jack functions), a value that is obtained somehow (such as checking the time in a Clock object), or some combination of these.

Parameters are the primary means of communicating with a function. A parameter can be used to do the following:

- Input information to a function (as in the jumping jacks)
- Provide output from the function (in this case, the parameter that passes must use a reference)

As a simple example, let's consider a function that computes the difference between two floating point numbers to give the change for a purchase. For example, given the purchase price and the amount tendered, the function computes the change. This function needs three values to operate:

- The value of the purchase price—`theprice` (given)
- The amount tendered—`theamount` (given)
- The change to be returned—`thechange` (computed by the function)

You can write a function that takes three parameters like these, computes the difference between the first two, and places the result in the third parameter. Remember that this third parameter must be passed as a reference. The first two parameters are used as input to the function and the third one is used as output.

This function does not request any input from the keyboard, nor does it write any output to the screen. However, the values of the purchase price and the amount tendered are provided as input to the function (not to the computer) by means of the first two parameters, `theprice` and `theamount`. In a similar manner, the result obtained is passed back to the calling program by means of the third parameter, `thechange`.

Input from the keyboard and output to the screen are done in the `mainprog` function.

In C++, you can implement this procedure with the code below:

```
void dif(float theprice,float theamount, float & thechange)
{
  thechange=theamount-theprice;
}
```

In this case, if you write a program that has values in `amount` and `price`, you can give the change by doing the following:

```
...
dif(price, amount, &change);
Cout<<change;
...
```

There is another way to communicate the result of a function so that the result can be used immediately in any expression. For example, you can do the following:

```
Cout<<dif(price,amount,&change);
```

This syntax is valid if the function returns a value as a result. Some programming languages differentiate between functions that return a result and those that do not. For example, in Pascal they are called functions if they return values; if they do not return values, they are called procedures.

How does the compiler know which functions return values and which do not? Why can't we issue a statement such as the following:

```
Number=JumpJack(Sal);
```

but we *can* issue a statement such as this one:

```
Price=ask("Please enter the price:");
```
What makes these functions different from each other?

Types of Return Values

In C++, the difference can be found in the function header—the first line, in which you give the name of the function. For example:

```
void dif(float theprice,float theamount, float & thechange)
```

The first thing that you write when defining a function is the type of the return value, if any. Do you remember that all our functions so far have had `void` preceding their names? This keyword meant that the function did not return a value.

To return a value, you must do the following:

- Specify the return type in the function header. A return type other than `void` will make the function return a result. If you omit the return type, C++ will assume you are returning an `int`.
- Include at least one statement in the function's body that specifies the result you want to return. This is done by using the keyword `return`.

For example, consider the simple case in which we want a modified `dif` function to return the difference between two floating point values that are passed as arguments. Since we want this function to return a floating point value as a result, the function header can be the following:

```
float dif(float value1, float value2)
```

There is nothing new about the syntax. This header tells the compiler that the function whose name is `dif` will take two arguments, `value1` and `value2` (both floating point numbers) and return a result that is also a floating point number. You no longer need a third parameter in which to store the result.

To finish the function's body, all we have to know is the syntax of the return. This is simple—it consists of the keyword `return`, followed by an expression:

```
return   expression ;
```
The expression is evaluated, and the result is returned by the function.

This example could then include the following function:

```
float dif(float value1, float value2)
{
   return value1-value2;
}
```

If a return is executed, the function ends! No other statements in the function are executed after the return.

This function can be used as illustrated in the program `c2change.cpp` below. In this program, we request that the user input a price to be paid and the amount tendered. The program then computes the change.

This is an extremely simple application of a function that returns a value. In fact, you could do better without using a function by computing the difference in the program itself. However, we will take advantage of the simplicity of this application to further our knowledge of functions.

The algorithm for this program is quite simple:

- Declare the variables, `price`, `amount`, and `change`
- Request that the user enter the purchase price
- Request that the user enter the amount tendered
- Compute the change (using the function `dif`)

- Inform the user of the value of the change

Here is the program `c2change.cpp` (as illustrated in Figure 6.1):

```
#include "franca.h"           // c2change.cpp
float dif(float value1,float value2)
{
  return value1-value2;
}
void mainprog()
{
  float price, amount,change;
  Box given("Amount:"),thechange("Change:");
  price=ask("What is the price to pay?");
  amount=ask("How much are you giving me?");
  change=dif(amount,price);
  given.say(amount);
  thechange.say(change);
}
```

Notice that after we **asked for** the values for `price` and `amount`, we used the expression:

```
change = dif (amount, price);
```

The value of this expression is obtained by the function `dif`. As we know, this function returns a value that is the difference between the first and second arguments. This returned value is then assigned to the variable `change`.

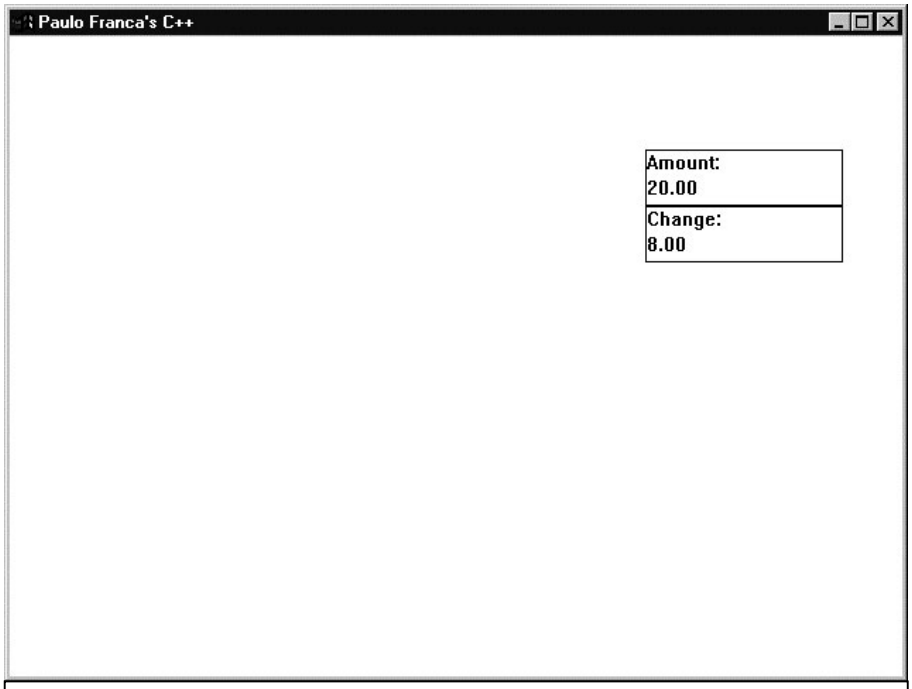


Fig 6.1 The result of c2change.cpp 1

In this expression, we use the arguments `amount` and `price`, in that order. The function `dif` uses the parameters `value1` and `value2`. When the program is executed and `dif` is invoked, the numeric value contained in the variable `amount` is copied to the variable `value1` in the function. In a similar manner, the value in `price` is copied to `value2`. The resulting values will then be used, and the result will be returned.

Remember that the correspondence between arguments is given solely by their order. If the expression were written as follows:

```
change = dif(price, amount);
```

the result would be completely wrong!

Originals and Copies

When you invoke a function that takes arguments, the original argument is not used. Instead, the function uses a copy of this variable (or object).

To use another example, modify the function `dif` so that the value in one of the arguments changes. For example, do the following:

```
float dif(float value1,float value2)
{
  value1=value1-value2;
  Cout<<value1;
  return value1;
}
```

In this case, we store the difference in `value1`, which is one of the arguments. `Cout` may be used to show the result that is stored in this variable. Still, the result is the same, because the function returns `value1`, which contains the difference.

If you include an output to display the value of the amount in the program, you will notice that this value was not changed.

This output was included for teaching purposes only. In this example, `mainprog` uses this result and could possibly display it. Avoid unnecessary output in functions—it may limit the reusability of your functions.

Inline Functions

Every time your program calls a function, the computer spends some time to switch to the function and then back to the program. Most likely, the values in the arguments must be copied to the function's parameters. This consumes time.

While using different programming languages, it is not generally advisable to design functions that perform extremely simple tasks, because the time the computer spends switching back and forth offsets the benefits of using a function.

However, C++ has a distinct feature that enables you to handle this kind of function: *Inline functions* are functions whose code is placed right where the function was invoked. In other words, the compiler actually writes a copy of your instructions in every place that you call the function. There is no switching back and forth at the point in which you invoked the function. On the other hand, unlike with ordinary functions, the code will be repeated every time the function is called.

Not all functions can become inline functions, though. There are a few restrictions—the most important one is that you cannot have repetitions inside an inline function (repetitions will be studied in SKILL seven).

Include the keyword `inline` in the function header to designate an inline function. Keep in mind, though, that this is strictly a C++ feature and is not available in other programming languages.

I will give an example of using an inline function in the next section of this skill, in which we reexamine the problems seen in "The Great Escape—Part 1" in SKILL two and use functions to simplify the solution.

The Great Escape—Part 2

Functions greatly simplify the task of having the robot move to draw squares. Notice how much easier this task becomes if the robot already knows how to draw a line!

If a function that makes the robot draw a line is available, the problem of drawing a square can be summarized as follows:

draw a line
turn right
draw a line
turn right
draw a line
turn right
draw a line
turn right

The last turn is not really necessary, but it is convenient, because you leave the robot facing the initial direction. You can use our robot Tracer to draw lines and squares.

Since we are not dealing with repetitions yet, we can think in terms of lines that are always three steps long. The algorithm is obvious:

This is how you draw a line of size 3:

mark
step
mark
step

Since the robot can only mark the square ahead of her (and not the one she is in), the total length of the line, including the initial square and the last one, is three squares. Again, this leaves the first square blank. If this bothers you, you can always correct the situation:

This is how you draw a line of size 3, including the first square:

mark
step
right
right // Turn back
mark // Mark first square
right // Turn back
right
mark
step

It is easy to write a function that implements any of the algorithms above:

```
Robot Tracer;
void line3()
{
   Tracer.mark();
   Tracer.step();
   Tracer.mark();
   Tracer.step();
}
```

Nevertheless, when you are designing a function, try to make it as useful as possible. That way, you may be able to reuse it someday.

This function would be a lot more useful if you could specify the line size. This will be done later when you learn how to handle repetitions. Still, you can add a little more versatility to this function if you allow the color to be chosen by means of a parameter. For example:

```
void line3(int color)
{
   Tracer.mark(color);
   Tracer.step();
   Tracer.mark(color);
   Tracer.step();
}
```

Does it bother you that you have to specify a color every time you invoke the function? No problem! You can specify a default value for the color in the function header:

```
void line3(int color=2)
```

Once you have written the line function, you can easily draw a square or, for that matter, you can consider writing a function that draws squares, as well. By doing so, you collect solutions to usual problems that you may face someday.

One possible implementation of the square function looks like the following:

```
void square3(int color=2)
{
  line3(color);
  Tracer.right();
  line3(color);
  Tracer.right();
  line3(color);
  Tracer.right();
  line3(color);
  Tracer.right();
}
```

When to Use a Function

To solve this simple square-drawing problem, functions can be used in several ways:

- Use two functions, `line3` and `square3`. `square3` draws a square by using another function, `line3`, which draws a line.
- Use three functions, `paint`, `line3`, and `square3`. You can use a function to mark and step to each square, because each time you want to paint a square, you have to mark and then step. This simplifies the code used for the `line3` function.
- Don't use the `square3` function. Use `line3` to draw the square in the program.
- Don't use the `line3` function. Use `square3`, which draws each line.
- Don't use functions at all. Write all the instructions to draw the square in the `mainprog` function.

Which Alternative Is Best?

There is no simple answer to this question. Several factors need to be considered to determine which alternative is best, and this may even vary from one person to another.

Clearly, the last alternative—not using functions at all—makes the program more difficult. Programmers who don't use functions typically spend more time developing and debugging, because of the following:

- The code ends up being more complex.
- No pieces were reused from previously developed software.

Some programmers maintain that using fewer functions makes their programs run faster. In general, this is a true statement. Each time a function is called, the computer spends additional time. However, not that many applications are so time-critical. In many situations, it may be preferable to finish your programs earlier and be able to change them easily upon request, instead

of ending up with a high- speed application that is difficult to change and that requires a long time to get ready.

The second alternative—using three levels of functions—may seem a little extreme. Functions that perform tiny tasks can become superfluous and steal time from your computer. This may or may not always be the case. But as far as you, the programmer, are concerned, this solution requires you to write the fewest lines of code! This makes it the ideal solution for the lazy programmer.

Is there anything wrong with being a lazy programmer? No, not at all! As long as you solve your problems while writing as little code as possible, you will actually climb the ladder of success. What is wrong with this solution then?

As I said before, it is questionable to use a function that is called a million times to do a puny job, because a lot more time is lost while switching than is used to do actual work. This leads to the natural conclusion that using functions in these cases should be avoided.

This is true in most programming languages, but is not completely true in C++—remember the inline functions?

Small pieces of program that do not contain a repetition and that are needed many times in the program are good candidates to become inline functions. However, remember that inline functions are specific to C++ and are not available in other languages. It remains a good practice to avoid using tiny functions!

The program `c2squar2.cpp` illustrates a solution that uses all three functions. In the `mainprog` function, the robot is instructed to take five steps toward the east then seven toward the south to start the drawing. Two squares are then built in different colors.

```
#include "franca.h"    //c2squar2.cpp
Robot Tracer;
inline void paint(int color=2)
{
  Tracer.mark(color);
  Tracer.step();
}
void line3 (int color=2)
{
  paint(color);
  paint(color);
}
void square3(int color=2)
{
  line3(color);
  Tracer.right();
  line3(color);
  Tracer.right();
  line3(color);
  Tracer.right();
  line3(color);
  Tracer.right();
}

void mainprog()
{
  Tracer.face(3);
  Tracer.step(5);
  Tracer.right();
  Tracer.step(7);
  Tracer.left();
  square3();
  Tracer.step(5);
  square3(3);
}
```

Rules and Regulations

Let's now summarize the syntax rules for the material covered in this skill.

Function definition

The *function definition* is a piece of program that explains the actions to be performed each time you invoke the function. The definition includes a function header and the function body. In general, a function will look like the following:

```
function header
{
    function body
}
```

Function header

The *function header* contains the following:

- The return type (functions that do not return a result use the type `void`)
- The function name (or identifier), and a list of parameters and their types enclosed in parentheses (if there are no parameters, the space inside the parentheses will be empty)

You may also see the keyword `void` inside the parentheses. For example, `float time(void)` is exactly the same as an empty parentheses—
`float time()`.

If there is more than one parameter, the parameters are separated by commas and each one must be preceded by its type or class. The parameters are already declared in the list itself.

For example, we have the following:

```
void JmpJack( athlete somebody)
```

This is a header for a function that:

- Does not return a result (`void`)
- Has a name (`JmpJack`)
- Takes one parameter of type `athlete`. This parameter will be known as `somebody` in the function. (Do not declare `somebody` to be of type `athlete` again in the function)

Function body

The *function body* contains the code that is executed when the function is called. This piece of program consists of one or more statements, which are placed inside braces.

For example, in the `JmpJack` function, the body is as follows:

```
somebody.up();
somebody.ready();
```

Therefore, the complete function is as follows:

```
void JumpJack( athlete somebody)
{
    somebody.up();
    somebody.ready();
}
```

Now, observe the following:

- If some arguments have default values and others do not, those with default values must be shown last in the argument list. Values assigned to default arguments must be in a valid scope.
- A function cannot be called before it has been defined!

Type matching

The *types* (or classes) of arguments must match those of the corresponding parameters when calling a function. If a function is defined as follows:

```
void jumps (Clock timekeeper, athlete somebody)
```

this function must be called using two arguments: the first must be `Clock` and the second must be `athlete`. The compiler checks whether this correspondence is done correctly and issues an error message if an error is found.

Function overloading

Unlike many other programming languages, C++ allows for more than one function to have the same identifier. This can be done if either the number or the type of arguments is not the same. We call this *function overloading*.

For example, a function `JmpJack` with one argument and another with two arguments can both be included in the same program. However, no other function `JmpJack` with one or two arguments of class `athlete` can be included.

The compiler will be able to differentiate between calls of the form:

```
JmpJack(Sal);
```

and those of the form:

```
JmpJack(Sal, Sally);
```

and will use the appropriate function.

Value vs. reference

By default, arguments are passed to a function *by value*. This means that a copy of the object or variable is made and is then used by the function. You can use and modify the parameter at will, without causing any modification to the original argument.

By including the symbol `&` before the parameter identifier, you indicate that all operations stated in the function involving that parameter are to use the original argument, instead of a copy.

Passing *by reference* can speed up your program if the object passed is complex. This will save the computer the time it needed to copy the object. It is not possible to pass a constant (e.g., a number) when the function expects a reference. For example, a function defined as follows:

```
float dif (float &value1,float &value2)
```

cannot be invoked as:

```
change = dif(100.00 ,- price);
```

The reference would allow the function to modify the value of `value1` and, in this case, the constant, `100.00`, is not supposed to be altered. This would result in an error message.

Function call

When you want to perform the set of actions you defined in a function, you can use a *function call* in your program. The function call consists of the function name followed by parentheses:

```
function_name ( argument_list )
```

If the function takes arguments, you must include the objects that will be used inside the parentheses. If the function takes more than one argument, the correspondence between the actual object and the parameter is established solely by the order in which they appear.

Function prototype

There may be situations in which you need to use a function before the function is defined. Since the compiler examines the source code sequentially, it will think you are calling a function that does not exist. This is not a very common situation. However, if you are faced with this problem, you will have to notify the compiler that there is a function with a given name that takes certain parameters and returns a result of a certain type. This is the *function prototype*.

The function prototype looks very much like the function header, except for the following:

- There is a semicolon at the end:

```
float change (float price, float amount);
```

- You may omit the parameter names:

```
float change (float, float);
```

It is a common mistake to include a semicolon after the function header—the compiler thinks this header is simply a prototype and never finds the function definition.

Function prototypes are not required unless you need to tell the compiler about a function that will be defined later in the code—but it is not a mistake to include them even when not needed.

Avoiding Multiple Inclusions of Header Files

When you use `include` to include files in your program, you may inadvertently copy the same file more than once. This may happen because either you have more than one `include` in your program or you included a file that also includes another file that was already requested.

For example, suppose you have a program that includes a program named `gymnast.h` as well as including `JumpJack.h`. As illustrated below, in this case the `gymnast.h` program also has an `include` for `JumpJack.h`.

Your program:

```
#include "gymnast.h"
#include "JumpJack.h"
```

The `gymnast.h` program:

```
#include "JumpJack.h"
```

If you want to do things right, you can prevent more than one copy from being included in your program by adding a few more lines to your header file. Add the following two lines at the beginning:

```
#ifndef     JUMPJACK_H
#define     JUMPJACK_H
```

and add this line at the end:

```
#endif
```

`JUMPJACK_H` should be substituted with a different name for each file. You can adopt the practice of using the name of the header file (`JumpJack`) in all uppercased letters, followed by an underscore and an uppercased *H*.

Statements that start with the character # in the first position are not specifically part of the C++ language. They are instructions for the part of the compiler called the preprocessor. These statements are also called preprocessor directives.

Try your hand at preprocessor syntax by writing a piece of code that uses an `include` to include the `jumpjack.h` file twice. Then, include the preprocessor directives given above in the `jumpjack.h` file and execute the same program.

When you've mastered these tasks, write a header file that contains functions for jumping jacks and leftrights. These functions should be able to receive an athlete as a parameter. Call this header file `fitness.h` and store it in your computer.

You may also want to write a program that uses the header file `fitness.h` and two athletes, Sal and Sally. This program should make Sal and Sally perform four jumping jacks and three leftrights.

Integers and Floating Point Numbers

An integer variable can store an integer number ranging from –32,768 to +32,767. Declare these variables by preceding the variable identifier with the keyword `int`. Numbers with a decimal part can be stored in variables of type `float`. Declare these by preceding the variable identifier with the keyword `float`. More than one variable of the same type can be declared in a statement. For example:

```
int number, alpha, count;
float x,y,z;
```

It is possible to set an initial value for the variable at the same time you declare it For example:

```
int one=1,two =2, three=3, other;
float x=3.45,y,z;
```

declares four variables: `one`, `two`, and `three` will be initialized to 1, 2, and 3 respectively, and `other` will not be initialized. Likewise, the floating variable `x` is initialized to the value 3.45. The variables `y` and `z` are declared, but are not initialized to any specific value.

You can change the value of the variable at any point in the program by showing the variable identifier on the left side of an assignment operator (=) and presenting an expression on the right side. In this case, the value of the expression substitutes the previous value of the variable. For example:

```
one=54;
two=one+three;
```

The left side of the assignment operator must designate a unique object or variable. It is invalid to have an expression such as the following:

```
one+two = three;
```

It is possible to use a `float` variable to assign a value to an `int` variable and vice versa. When a `float` value is assigned to an integer, the fractional part is lost (it is truncated, not rounded). When an integer value is assigned to a `float` variable, a fractional part consisting of zeros is assumed.

Expressions

Expressions are sequences containing variable identifiers, arithmetic operators, function calls, parentheses, and constants (numbers) arranged according to predetermined rules. Expressions indicate how to operate with the variables and constants to produce a result.

The arithmetic operators are as follows:

+	addition
–	subtraction
*	multiplication
/	division
%	remainder

In addition, the assignment operator (=) can also be used in arithmetic expressions.

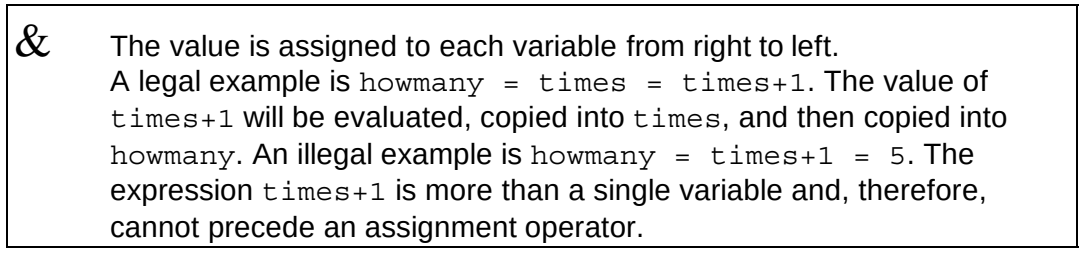
Definitions

- Operators can be +, –, *, /, %, and =.
- A term is a variable (or object), a constant, or a function call.

Rules

- Terms must be separated by an operator.
- The two terms operate according to the rule prescribed by that operator.

- Operations are executed according to their priority. Highest priority operations use *, /, and %. Lowest priority operations use + and –. Operations of the same priority are evaluated in the order they are written (left to right).
- A term may be preceded by the operator –.
  - The negative of the value will be taken. (It is also possible to precede a term by the operator +.)
- Pairs of parentheses may be included anywhere to specify priority.
  - Expressions inside parentheses will be evaluated first. If more than one pair of parentheses is included, the inner parentheses will be evaluated first.
- The increment (++) and decrement (—) operators may be used immediately before or immediately after a variable.
  - The variable will be increment or decremented.
- A variable (or object) may be used on the left side of an assignment operator (=).
  - The result of the expression is assigned to the variable whose identifier is on the left side of the =.
  - More than one assignment operator may be used in an expression, provided each one is preceded by only a variable.



Try These for Fun…

- Write a function `symmetry (athlete he, athlete she);` that makes two athletes perform an exercise simultaneously. However, the exercises should be performed symmetrically, in the following manner:
  - `he` should do the sequence:

    ready
    up
    left
    up
    right
    up
  - `she` should do the sequence:

ready
up
right
up
left
up

- Write a program that makes the two athletes exercise six times.

Are You Experienced?

Now you can…

Return values as results of functions

Use inline functions

Use functions to solve your problems