SESSION

9

Functions

Session Checklist

- ✔ Writing void functions
- ✔ Writing functions with multiple arguments
- ✔ Overloading functions
- ✔ Creating function templates
- ✔ Determining variable storage class

30 Min. To Go

Some of the example programs in Session 8 are already becoming a bit involved and we have a lot more C++ to learn. Programs with multiple levels of nesting of flow control can be difficult to follow. Add the numerous and complicated branching that a "real-world" application requires and programs would become virtually impossible to follow.

Fortunately, C++ provides a means for separating a stand-alone block of code into a stand-alone entity known as a function. In this Session, I delve into how to declare, create, and use C++ functions.

Sample Function Code

Like so many things, functions are best understood by example. This section starts with an example program, FunctionDemo, which simplifies the NestedDemo program in Session 8 by defining a function to contain part of the logic. This section then explains how the function is defined and how it is invoked using the example code as a pattern.

Sample code

The NestedDemo in Session 8 involves an inner loop that accumulates a sequence of numbers and an outer loop that repeats the process until the user decides to quit. Logically we could separate the inner loop, the part that adds a sequence of numbers, from the outer loop that repeats the process.

The following code example shows the NestedDemo is simplified by creating the function `sumSequence()`.

Tip

The names of functions are normally written with a set of parentheses immediately following.

```
// FunctionDemo - demonstrate the use of functions
//                 by breaking the inner loop of the
//                 NestedDemo program off into its own
//                 function

#include <stdio.h>
#include <iostream.h>

// sumSequence - add a sequence of numbers entered from
//                the keyboard until the user enters a
//                negative number.
//                return - the summation of numbers entered
int sumSequence(void)
{
    // loop forever
    int nAccumulator = 0;
    for(;;)
     {
        // fetch another number
        int nValue = 0;
        cout << “Enter next number: “;
        cin  > nValue;

        // if it’s negative...
        if (nValue < 0)
        {
            // ...then exit from the loop
            break;
        }

        // ...otherwise add the number to the
        // accumulator
        nAccumulator = nAccumulator + nValue;
    }

    // return the accumulated value
    return nAccumulator;
}

int main(int nArg, char* pszArgs[])
{                                       // Begin Main
    cout << “This program sums multiple series\n”
         << “of numbers. Terminate each sequence\n”
         << “by entering a negative number.\n”
         << “Terminate the series by entering two\n”
         << “negative numbers in a row\n”;

    // accumulate sequences of numbers...
    int nAccumulatedValue;
    do
    {
        // sum a sequence of numbers entered from
        // the keyboard
        cout << “\nEnter next sequence\n”;
        nAccumulatedValue = sumSequence();

        // now output the accumulated result
        cout << “\nThe total is “
```

```
             << nAccumulatedValue
             << “\n”;

    // ...until the sum returned is 0
    } while (nAccumulatedValue != 0);
    cout << “Program terminating\n”;
    return 0;
}                                       // End Main
```

Calling the function sumSequence()

Let’s first concentrate on the main program contained within the open and closed braces marked with the comments `Begin Main` and `End Main`. This section of code looks identical to programs that we wrote previously.

The line

```
nAccumulatedValue = sumSequence();
```

calls the function `sumSequence()` and stores the value returned in the variable `nAccumulatedValue`. This value is subsequently output to the standard output on the following three lines. The main program continues to loop until the sum returned by the inner function is 0, indicating that the user has finished calculating sums.

Defining the sumSequence() *function*

The block of code that begins on line 13 and continues through line 38 makes up the function `sumSequence()`.

When the main program invokes the function `sumSequence()` on line 55, control passes from the call to the beginning of the function on line 14. Program execution continues from that point.

Lines 16 through 34 are identical to that found in the inner loop of NestedDemo. After the program exits that loop, control passes to the return statement on line 37. At this point, control passes back to the call statement on line 55 along with the value contained in `nAccumulator`. On line 55, the main program stores the `int` returned in the local variable `nAccumulatedValue` and continues execution.

20 Min.
To Go

Note

In this case, the call to `sumSequence()` is an expression, because it has a value. Such a call can be used anywhere an expression is allowed.

Part II–Saturday Morning
Session 9

Function

The FunctionDemo program demonstrates the definition and use of a simple function.

SYNTAX

A **function** is a logically separate block of C++ code. The function construct has the form:

```
<return type> name(<arguments to the function>)
{
    // ...
    return <expression>;
}
```

The arguments to a function are values that can be passed for the function to use as input. The return value is a value that the function returns. For example, in the call to the function `square(10)`, the value 10 is an argument to the function `square()`. The returned value is 100.

Both the arguments and the return value are optional. If either is absent, the keyword `void` is used instead. That is, if a function has a void argument list, the function does not take any arguments when called. If the return type is void, then the function does not return a value to the caller.

In the example FunctionDemo program, the name of the function is `sumSequence()`, the return type is `int`, and there are no arguments.

Tip

The default argument type to a function is `void`. Thus, a function `int fn(void)` may be declared as `int fn()`.

Why use functions?

The advantage of a function over other C++ control commands is that it cordons off a set of code with a particular purpose from other code in the program. By being separate, the programmer can concentrate on the one function when writing it.

Tip

A good function can be described using a single sentence that contains a minimum number of ors and ands. For example, "the function `sumSequence` accumulates a sequence of integer values entered by the user." This definition is concise and clear.

The function construct made it possible for me to write essentially two distinct parts of the FunctionDemo program. I concentrated on creating the sum of a sequence of numbers when writing the `sumSequence()` function. I didn't think at all about other code that might call the function.

Likewise, when writing `main()`, I could concentrate on handling the summation returned by `sumSequence()`, while thinking only of what the function did, and not about how it worked.

Simple functions

The simple function `sumSequence()` returns an integer value which it calculates. Functions may return any of the regular types of variables. For example, a function might return a `double` or a `char`.

If a function returns no value, then the return type of the function is labeled `void`.

A function may be labeled by its return type. Thus, a function that returns an `int` is often known as an *integer* function. A function that returns no value is known as a *void* function.

For example, the following void function performs an operation but returns no value:

```
void echoSquare()
{
    int nValue;
    cout << “Enter a value:”;
cin > nValue;
cout << “\n The square is:” << nValue * nValue << “\n”;
    return;
}
```

Control begins at the open brace and continues through to the `return` statement. The `return` statement in a void function is not followed by a value.

The `return` statement in a void function is optional. If it is not present, then execution returns to the calling function when control encounters the close brace.

Functions with arguments

Simple functions are of limited use because the communication from such functions is one-way, through the return value. Two-way communication is preferred. Such communication is through function arguments. A *function argument* is a variable whose value is passed to the calling function during the call operation.

Example function with arguments

The following example defines and uses a `square()` function which returns the square of a double-precision float passed to it:

```
// SquareDemo - demonstrate the use of a function
//              which processes arguments

#include <stdio.h>
#include <iostream.h>

// square - returns the square of its argument
//          dVar - the value to be squared
//          returns - sqare of dVar
double square(double dVar)
{
    return dVar * dVar;
}

int sumSequence(void)
{
    // loop forever
    int nAccumulator = 0;
     for(;;)
     {
        // fetch another number
        double dValue = 0;
        cout << “Enter next number: “;
        cin  > dValue;

        // if it’s negative...

        if (dValue < 0)
        {
```

Part II–Saturday Morning
Session 9

```
            // ...then exit from the loop
            break;
        }

        // ...otherwise calculate the square
        int nValue = (int)square(dValue);

        // now add the square to the
        // accumulator
        nAccumulator = nAccumulator + nValue;
    }

    // return the accumulated value
    return nAccumulator;
}

int main(int nArg, char* pszArgs[])
{
    cout << “This program sums multiple series\n”
         << “of numbers. Terminate each sequence\n”
         << “by entering a negative number.\n”
         << “Terminate the series by entering two\n”
         << “negative numbers in a row\n”;

    // Continue to accumulate numbers...
    int nAccumulatedValue;
    do
    {
        // sum a sequence of numbers entered from
        // the keyboard
        cout << “\nEnter next sequence\n”;
        nAccumulatedValue = sumSequence();

        // now output the accumulated result
        cout << “\nThe total is “
             << nAccumulatedValue
             << “\n”;

    // ...until the sum returned is 0
    } while (nAccumulatedValue != 0);
```

```
        cout << "Program terminating\n";
        return 0;
    }
```

This is the same FunctionDemo program except that SquareDemo adds the square of the values entered.

The value of `dValue` is passed to the function `square()` on the line

```
int nValue = (int)square(dValue);
```

contained within the function `sumSequence()`. The function `square()` multiplies the value passed to it on line 12 and returns the result. The returned value is stored in the variable `dSquare`, which is added to the accumulator value on line 38.

Functions with multiple arguments

Functions may have multiple arguments separated by commas. The following function returns the product of its two arguments:

```
int product(int nArg1, int nArg2)
{
    return nArg1 * nArg2;
}
```

Casting Values

Line 38 of the SquareDemo program contains an operator that we have not seen before:

```
nAccumulator = nAccumulator + (int)dValue;
```

The `(int)` in front of the `dValue` indicates that the programmer wants to convert the `dValue` variable from its current type, in this case `double`, into an `int`.

A *cast* is an explicit conversion from one type to another. Any numeric type may be cast into any other numeric type. Without such a cast, C++ would have converted the types anyway, but would have generated a warning.

main() exposed

It should be clear that `main()` is nothing more than a function, albeit, a function with strange arguments.

When a program is built, C++ adds some boilerplate code that executes before your program ever starts. This code sets up the environment in which your program operates. For example, this boilerplate code opens the `cin` and `cout` input and output objects.

Once the environment has been established, the C++ boilerplate code calls the function `main()` thereby beginning execution of your code. When your program is complete, it exits from `main()`. This enables the C++ boilerplate to clean up a few things before turning over control to the operating system which kills the program.

10 Min. To Go

Multiple functions with the same nickname

Two functions in a single program cannot share the same name. If they did, C++ would have no way to distinguish them when they are called. However, in C++ the name of the function includes the number and type of its arguments. Thus, the following are not the same functions:

```
void someFunction(void)
{
    // ....perform some function
}
void someFunction(int n)
{
    // ...perform some different function
}
void someFunction(double d)
{
    // ...perform some very different function
}
void someFunction(int n1, int n2)
{
    // ....do something different yet
}
```

`someFunction()` is a shorthand name, or nickname, for both functions, in the same way that Stephen is shorthand for my name. Whether I use the shorthand name when describing the function or not, C++ still knows the functions as someFunction(void), someFunction(int), someFunction(double), and someFunction(int, int).

Note

void as an argument type is optional; sumFunction(void) and sumFunction() are the same function.

A typical application might appear as follows:

```
int nVariable1, nVariable2;    // equivalent to
                               // int Variable1;
                               // int Variable2;
double dVariable;

// functions are distinguished by the type of
// the argument passed
someFunction();                 // calls someFunction(void)
someFunction(nVariable1);       // calls someFunction(int)
someFunction(dVariable);        // calls someFunction(double)
someFunction(nVariable1, nVariable2); // calls
                                // someFunction(int, int)

// this works for constants as well
someFunction(1);                // calls someFunction(int)
someFunction(1.0);              // calls someFunction(double)
someFunction(1, 2);             // calls someFunction(int, int)
```

In each case, the type of the arguments matches the full name of the three functions.

Note

The return type is not part of the name of the function. Thus, the following two functions have the same name and cannot be part of the same program:

```
int someFunction(int n);      // full name of the function
                              // is someFunction(int)
double someFunction(int n);   // same name
```

Function Prototypes

In the example programs so far, the target functions sumSequence() and square() were both defined in code that appeared before the actual call. This doesn't have to be the case: a function may be defined anywhere in the module. A *module* is another name for a C++ source file.

However, something has to tell main() the full name of the function before it can be called. Consider the following code snippet:

```
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}
int someFunc(double dArg1, int nArg2)
{
    // ...do something
}
```

The call to someFunc() from within main() doesn't know the full name of the function. It might surmise from the arguments that the name is someFunc(int, int) and that its return type is void; however, as you can see, this is wrong.

What is needed is some way to inform main() of the full name of someFunc() before it is used. What is needed is a function declaration. A *function declaration* or *prototype* appears the same as a function with no body. In use, a prototype appears as follows:

```
int someFunc(double, int);
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}
int someFunc(double dArg1, int nArg2)
{
    // ...do something
}
```

The call in `main()` now knows to cast the 1 to a double before making the call. In addition, `main()` knows that the function `someFunc()` returns an `int`; however, `main()` doesn't do anything with the result.

Tip

C++ allows the programmer to throw away the value returned by a function.

Variable Storage Types

There are three different places that function variables can be stored. Variables declared within a function are said to be local. In the following example, the variable `nLocal` is local to the function `fn()`:

```
int nGlobal;
void fn()
{
    int nLocal;
    static int nStatic;
}
```

The variable `nLocal` doesn't "exist" until the function `fn()` is called. In addition, only `fn()` has access to `nLocal` — other functions cannot "reach into" the function to access it.

By comparison, the variable `nGlobal` exists as long as the program is running. All variables have access to `nGlobal` all of the time.

The static variable `nStatic` is something of a mix between a local and a global variable. The variable `nStatic` is created when execution first reaches the declaration (roughly, when the function `fn()` is called). In addition, `nStatic` is only accessible within `fn()`. Unlike `nLocal`, however, `nStatic` continues to exist even after the program exits `fn()`. If `fn()` assigns a value to `nStatic` once, it will still be there the next time that `fn()` is called.

Done!

Note

There is a fourth variable type, `auto`, but today it has the same meaning as `local`.

REVIEW

By this time, you should have some idea of how complex a program can become and how creating small functions can simplify program logic. Well-conceived functions are small, ideally less than 50 lines and with fewer than 7 `if` or looping commands. Such functions are easier to grasp and, therefore, easier to write and debug. After all, isn't that the idea?

- C++ functions are a means for dividing code into bite-sized chunks.
- Functions may have any number of arguments, which are values passed to the function.
- Functions may return a single value to the caller.
- Function names may be overloaded as long as they can be distinguished by the number and types of their arguments.

It's all very nice to outfit your programs with numerous expressions in multiple variables that are divided into functions, but it isn't of much use if you can't get your programs to work. In Session 10, you will see some of the most basic techniques for finding coding errors in your programs.

QUIZ YOURSELF

1. How do you call a function? (See "Calling the Function sumSequence().")
2. What does a return type of `void` mean? (See "Function.")
3. Why write functions? (See "Why Use Functions?")
4. What is the difference between a local variable and a global variable? (See "Variable Storage Types.")