CHAPTER VIII

Functions

The focus of this chapter is functions—when to declare them, how to declare them, and different techniques for using them. Functions are the building blocks of the C language, and mastering functions is one of the key elements needed to be a successful C programmer.

When you read this chapter, keep in mind the functions you have written and whether or not you are squeezing every bit of efficiency out of them. If you aren't doing so, you should apply some of the techniques presented in this chapter to make your programs faster and more efficient. Also, keep in mind some of the tips presented here regarding good programming practice—perhaps you can improve your function-writing skills by examining some of the examples provided.

VIII.1: When should I declare a function?

Answer:

Functions that are used only in the current source file should be declared as `static` (see FAQ VIII.4), and the function's declaration should appear in the current source file along with the definition of the function. Functions used outside of the current source file should have their declarations put in a *header file*, which can be included in whatever

source file is going to use that function. For instance, if a function named `stat_func()` is used only in the source file stat.c, it should be declared as shown here:

```
/* stat.c */

#include <stdio.h>

static int stat_func(int, int);  /* static declaration of stat_func() */
➥void main(void);

void main(void)
{

     ...

     rc = stat_func(1, 2);

     ...

}

/* definition (body) of stat_func() */

static int stat_func(int arg1, int arg2)
{

     ...

     return rc;

}
```

In this example, the function named `stat_func()` is never used outside of the source file stat.c. There is therefore no reason for the prototype (or declaration) of the function to be visible outside of the stat.c source file. Thus, to avoid any confusion with other functions that might have the same name, the declaration of `stat_func()` should be put in the same source file as the declaration of `stat_func()`.

In the following example, the function `glob_func()` is declared and used in the source file global.c and is used in the source file extern.c. Because `glob_func()` is used outside of the source file in which it's declared, the declaration of `glob_func()` should be put in a header file (in this example, named proto.h) to be included in both the global.c and the extern.c source files. This is how it's done:

File: proto.h

```
/* proto.h */

int glob_func(int, int);  /* declaration of the glob_func() function */
```

File: global.c

```
/* global.c */

#include <stdio.h>
#include "proto.h"   /* include this file for the declaration of
                        glob_func() */

void main(void);

void main(void)
```

```
{

     ...

     rc = glob_func(1, 2);

     ...

}

/* definition (body) of the glob_func() function */

int glob_func(int arg1, int arg2)
{

     ...

     return rc;

}
```

File: extern.c

```
/* extern.c */

#include <stdio.h>
#include "proto.h"   /* include this file for the declaration of
                        glob_func() */

void ext_func(void);

void ext_func(void)
{

     ...

     /* call glob_func(), which is defined in the global.c source file */

     rc = glob_func(10, 20);

     ...

}
```

In the preceding example, the declaration of `glob_func()` is put in the header file named proto.h because `glob_func()` is used in both the global.c and the extern.c source files. Now, whenever `glob_func()` is going to be used, you simply need to include the proto.h header file, and you will automatically have the function's declaration. This will help your compiler when it is checking parameters and return values from global functions you are using in your programs. Notice that your function declarations should always appear *before the first function declaration in your source file.*

In general, if you think your function might be of some use outside of the current source file, you should put its declaration in a header file so that other modules can access it. Otherwise, if you are sure your function will never be used outside of the current source file, you should declare the function as `static` and include the declaration only in the current source file.

Cross Reference:

VIII.2: Why should I prototype a function?

VIII.3: How many parameters should a function have?

VIII.4: What is a static function?

VIII.2: Why should I prototype a function?

Answer:

A function prototype tells the compiler what kind of arguments a function is looking to receive and what kind of return value a function is going to give back. This approach helps the compiler ensure that calls to a function are made correctly and that no erroneous type conversions are taking place. For instance, consider the following prototype:

```
int some_func(int, char*, long);
```

Looking at this prototype, the compiler can check all references (including the definition of `some_func()`) to ensure that three parameters are used (an integer, a character pointer, and then a long integer) and that a return value of type integer is received. If the compiler finds differences between the prototype and calls to the function or the definition of the function, an error or a warning can be generated to avoid errors in your source code. For instance, the following examples would be flagged as incorrect, given the preceding prototype of `some_func()`:

```
x = some_func(1);                    /* not enough arguments passed */

x = some_func("HELLO!", 1, "DUDE!"); /* wrong type of arguments used */

x = some_func(1, str, 2879, "T");    /* too many arguments passed */
```

In the following example, the return value expected from `some_func()` is not an integer:

```
long* lValue;

lValue = some_func(1, str, 2879);    /* some_func() returns an int,
                                        not a long* */
```

Using prototypes, the compiler can also ensure that the function definition, or body, is correct and correlates with the prototype. For instance, the following definition of `some_func()` is not the same as its prototype, and it therefore would be flagged by the compiler:

```
int some_func(char* string, long lValue, int iValue)  /* wrong order of
                                                         parameters */
{

     ...

}
```

The bottom line on prototypes is that you should always include them in your source code because they provide a good error-checking mechanism to ensure that your functions are being used correctly. Besides, many of today's popular compilers give you warnings when compiling if they can't find a prototype for a function that is being referenced.

Cross Reference:

VIII.1: When should I declare a function?

VIII.3: How many parameters should a function have?

VIII.4: What is a static function?

VIII.3: How many parameters should a function have?

Answer:

There is no set number or "guideline" limit to the number of parameters your functions can have. However, it is considered bad programming style for your functions to contain an inordinately high (eight or more) number of parameters. The number of parameters a function has also directly affects the speed at which it is called—the more parameters, the slower the function call. Therefore, if possible, you should minimize the number of parameters you use in a function. If you are using more than four parameters, you might want to rethink your function design and calling conventions.

One technique that can be helpful if you find yourself with a large number of function parameters is to put your function parameters in a structure. Consider the following program, which contains a function named `print_report()` that uses 10 parameters. Instead of making an enormous function declaration and prototype, the `print_report()` function uses a structure to get its parameters:

```
#include <stdio.h>

typedef struct
{
     int       orientation;
     char      rpt_name[25];
     char      rpt_path[40];
     int       destination;
     char      output_file[25];
     int       starting_page;
     int       ending_page;
     char      db_name[25];
     char      db_path[40];
     int       draft_quality;
} RPT_PARMS;

void main(void);
int print_report(RPT_PARMS*);

void main(void)
{

     RPT_PARMS rpt_parm; /* define the report parameter
                            structure variable */

     ...

     /* set up the report parameter structure variable to pass to the
     print_report() function */
```

```
    rpt_parm.orientation = ORIENT_LANDSCAPE;
    rpt_parm.rpt_name = "QSALES.RPT";
    rpt_parm.rpt_path = "C:\REPORTS";
    rpt_parm.destination = DEST_FILE;
    rpt_parm.output_file = "QSALES.TXT";
    rpt_parm.starting_page = 1;
    rpt_parm.ending_page = RPT_END;
    rpt_parm.db_name = "SALES.DB";
    rpt_parm.db_path = "C:\DATA";
    rpt_parm.draft_quality = TRUE;

    /* Call the print_report() function, passing it a pointer to the
       parameters instead of passing it a long list of 10 separate
       parameters. */

    ret_code = print_report(&rpt_parm);

    ...

}

int print_report(RPT_PARMS* p)
{

    int rc;

    ...

    /* access the report parameters passed to the print_report()
       function */

    orient_printer(p->orientation);

    set_printer_quality((p->draft_quality == TRUE) ? DRAFT : NORMAL);

    ...

    return rc;

}
```

The preceding example avoided a large, messy function prototype and definition by setting up a predefined structure of type `RPT_PARMS` to hold the 10 parameters that were needed by the `print_report()` function. The only possible disadvantage to this approach is that by removing the parameters from the function definition, you are bypassing the compiler's capability to type-check each of the parameters for validity during the compile stage.

Generally, you should keep your functions small and focused, with as few parameters as possible to help with execution speed. If you find yourself writing lengthy functions with many parameters, maybe you should rethink your function design or consider using the structure-passing technique presented here. Additionally, keeping your functions small and focused will help when you are trying to isolate and fix bugs in your programs.

Cross Reference:

VIII.1: When should I declare a function?

VIII.2: Why should I prototype a function?

VIII.4: What is a static function?

VIII.4: What is a static function?

Answer:

A *static* function is a function whose *scope* is limited to the current source file. Scope refers to the visibility of a function or variable. If the function or variable is visible outside of the current source file, it is said to have global, or external, scope. If the function or variable is not visible outside of the current source file, it is said to have local, or static, scope.

A static function therefore can be seen and used only by other functions within the current source file. When you have a function that you know will not be used outside of the current source file or if you have a function that you do not want being used outside of the current source file, you should declare it as `static`. Declaring local functions as `static` is considered good programming practice. You should use static functions often to avoid possible conflicts with external functions that might have the same name.

For instance, consider the following example program, which contains two functions. The first function, `open_customer_table()`, is a global function that can be called by any module. The second function, `open_customer_indexes()`, is a local function that will never be called by another module. This is because you can't have the customer's index files open without first having the customer table open. Here is the code:

```
#include <stdio.h>

int open_customer_table(void);       /* global function, callable from
                                        any module */
static int open_customer_indexes(void); /* local function, used only in
                                           this module */

int open_customer_table(void)
{

     int ret_code;

     /* open the customer table */

     ...

     if (ret_code == OK)
     {
          ret_code = open_customer_indexes();
     }

     return ret_code;

}
```

```
static int open_customer_indexes(void)
{

     int ret_code;

     /* open the index files used for this table */

     ...

     return ret_code;

}
```

Generally, if the function you are writing will not be used outside of the current source file, you should declare it as `static`.

Cross Reference:

VIII.1: When should I declare a function?

VIII.2: Why should I prototype a function?

VIII.3: How many parameters should a function have?

VIII.5: Should a function contain a *return* statement if it does not return a value?

Answer:

In C, *void* functions (those that do not return a value to the calling function) are not required to include a `return` statement. Therefore, it is not necessary to include a `return` statement in your functions declared as being void.

In some cases, your function might trigger some critical error, and an immediate exit from the function might be necessary. In this case, it is perfectly acceptable to use a `return` statement to bypass the rest of the function's code. However, keep in mind that it is not considered good programming practice to litter your functions with `return` statements—generally, you should keep your function's exit point as focused and clean as possible.

Cross Reference:

VIII.8: What does a function declared as PASCAL do differently?

VIII.9: Is using `exit()` the same as using `return`?

VIII.6: How can you pass an array to a function by value?

Answer:

An array can be passed to a function by value by declaring in the called function the array name with square brackets ([and]) attached to the end. When calling the function, simply pass the address of the array (that is, the array's name) to the called function. For instance, the following program passes the array `x[]` to the function named `byval_func()` by value:

```
#include <stdio.h>

void byval_func(int[]);       /* the byval_func() function is passed an
                                 integer array by value */
void main(void);

void main(void)
{

     int x[10];
     int y;

     /* Set up the integer array. */

     for (y=0; y<10; y++)
          x[y] = y;

     /* Call byval_func(), passing the x array by value. */

     byval_func(x);

}

/* The byval_function receives an integer array by value. */

void byval_func(int i[])
{

     int y;

     /* Print the contents of the integer array. */

     for (y=0; y<10; y++)
          printf("%d\n", i[y]);

}
```

In this example program, an integer array named `x` is defined and initialized with 10 values. The function `byval_func()` is declared as follows:

```
int byval_func(int[]);
```

The `int[]` parameter tells the compiler that the `byval_func()` function will take one argument—an array of integers. When the `byval_func()` function is called, you pass the address of the array to `byval_func()`:

```
byval_func(x);
```

Because the array is being passed by value, an exact copy of the array is made and placed on the stack. The called function then receives this copy of the array and can print it. Because the array passed to `byval_func()` is a copy of the original array, modifying the array within the `byval_func()` function has no effect on the original array.

Passing arrays of any kind to functions can be very costly in several ways. First, this approach is very inefficient because an entire copy of the array must be made and placed on the stack. This takes up valuable program time, and your program execution time is degraded. Second, because a copy of the array is made, more memory (stack) space is required. Third, copying the array requires more code generated by the compiler, so your program is larger.

Instead of passing arrays to functions by value, you should consider passing arrays to functions by reference: this means including a pointer to the original array. When you use this method, no copy of the array is made. Your programs are therefore smaller and more efficient, and they take up less stack space. To pass an array by reference, you simply declare in the called function prototype a pointer to the data type you are holding in the array.

Consider the following program, which passes the same array (`x`) to a function:

```
#include <stdio.h>

void const_func(const int*);
void main(void);

void main(void)
{

     int x[10];
     int y;

     /* Set up the integer array. */

     for (y=0; y<10; y++)
          x[y] = y;

     /* Call const_func(), passing the x array by reference. */

     const_func(x);

}

/* The const_function receives an integer array by reference.
   Notice that the pointer is declared as const, which renders
   it unmodifiable by the const_func() function. */

void const_func(const int* i)
{

     int y;

     /* Print the contents of the integer array. */
```

```
    for (y=0; y<10; y++)
        printf("%d\n", *(i+y));

}
```

In the preceding example program, an integer array named `x` is defined and initialized with 10 values. The function `const_func()` is declared as follows:

```
int const_func(const int*);
```

The `const int*` parameter tells the compiler that the `const_func()` function will take one argument—a constant pointer to an integer. When the `const_func()` function is called, you pass the address of the array to `const_func()`:

```
const_func(x);
```

Because the array is being passed by reference, no copy of the array is made and placed on the stack. The called function receives simply a constant pointer to an integer. The called function must be coded to be smart enough to know that what it is really receiving is a constant pointer to an array of integers. The `const` modifier is used to prevent the `const_func()` from accidentally modifying any elements of the original array.

The only possible drawback to this alternative method of passing arrays is that the called function must be coded correctly to access the array—it is not readily apparent by the `const_func()` function prototype or definition that it is being passed a reference to an array of integers. You will find, however, that this method is much quicker and more efficient, and it is recommended when speed is of utmost importance.

Cross Reference:

VIII.8: What does a function declared as PASCAL do differently?

VIII.7: Is it possible to execute code even after the program exits the *main()* function?

Answer:

The standard C library provides a function named `atexit()` that can be used to perform "cleanup" operations when your program terminates. You can set up a set of functions you want to perform automatically when your program exits by passing function pointers to the `atexit()` function. Here's an example of a program that uses the `atexit()` function:

```
#include <stdio.h>
#include <stdlib.h>

void close_files(void);
void print_registration_message(void);
int main(int, char**);

int main(int argc, char** argv)
{
```

```
    ...

    atexit(print_registration_message);
    atexit(close_files);

    while (rec_count < max_records)
    {

         process_one_record();

    }

    exit(0);

}
```

This example program uses the `atexit()` function to signify that the `close_files()` function and the `print_registration_message()` function need to be called automatically when the program exits. When the `main()` function ends, these two functions will be called to close the files and print the registration message.

There are two things that should be noted regarding the `atexit()` function. First, the functions you specify to execute at program termination must be declared as void functions that take no parameters. Second, the functions you designate with the `atexit()` function are stacked in the order in which they are called with `atexit()`, and therefore they are executed in a last-in, first-out (LIFO) method. Keep this information in mind when using the `atexit()` function. In the preceding example, the `atexit()` function is stacked as shown here:

```
atexit(print_registration_message);
atexit(close_files);
```

Because the LIFO method is used, the `close_files()` function will be called first, and then the `print_registration_message()` function will be called.

The `atexit()` function can come in handy when you want to ensure that certain functions (such as closing your program's data files) are performed before your program terminates.

Cross Reference:

VIII.9: Is using `exit()` the same as using `return`?

VIII.8: What does a function declared as *PASCAL* do differently?

Answer:

A C function declared as `PASCAL` uses a different *calling convention* than a "regular" C function. Normally, C function parameters are passed right to left; with the `PASCAL` calling convention, the parameters are passed left to right.

Consider the following function, which is declared normally in a C program:

```
int regular_func(int, char*, long);
```

Using the standard C calling convention, the parameters are pushed on the stack from right to left. This means that when the `regular_func()` function is called in C, the stack will contain the following parameters:

```
long
char*
int
```

The function calling `regular_func()` is responsible for restoring the stack when `regular_func()` returns. When the PASCAL calling convention is being used, the parameters are pushed on the stack from left to right.

Consider the following function, which is declared as using the PASCAL calling convention:

```
int PASCAL pascal_func(int, char*, long);
```

When the function `pascal_func()` is called in C, the stack will contain the following parameters:

```
int
char*
long
```

The function being called is responsible for restoring the stack pointer. Why does this matter? Is there any benefit to using PASCAL functions?

Functions that use the PASCAL calling convention are more efficient than regular C functions—the function calls tend to be slightly faster. Microsoft Windows is an example of an operating environment that uses the PASCAL calling convention. The Windows SDK (Software Development Kit) contains hundreds of functions declared as PASCAL.

When Windows was first designed and written in the late 1980s, using the PASCAL modifier tended to make a noticeable difference in program execution speed. In today's world of fast machinery, the PASCAL modifier is much less of a catalyst when it comes to the speed of your programs. In fact, Microsoft has abandoned the PASCAL calling convention style for the Windows NT operating system.

In your world of programming, if milliseconds make a big difference in your programs, you might want to use the PASCAL modifier when declaring your functions. Most of the time, however, the difference in speed is hardly noticeable, and you would do just fine to use C's regular calling convention.

Cross Reference:

VIII.6: How can you pass an array to a function by value?

VIII.9: Is using *exit()* the same as using *return*?

Answer:

No. The `exit()` function is used to exit your program and return control to the operating system. The `return` statement is used to return from a function and return control to the calling function. If you issue a return from the `main()` function, you are essentially returning control to the calling function, which is the operating system. In this case, the `return` statement and `exit()` function are similar. Here is an example of a program that uses the `exit()` function and `return` statement:

```
#include <stdio.h>
#include <stdlib.h>

int main(int, char**);
```

```
int do_processing(void);
int do_something_daring();

int main(int argc, char** argv)
{

     int ret_code;

     if (argc < 3)
     {

          printf("Wrong number of arguments used!\n");

          /* return 1 to the operating system */

          exit(1);

     }

     ret_code = do_processing();

     ...

     /* return 0 to the operating system */

     exit(0);

}

int do_processing(void)
{

     int rc;

     rc = do_something_daring();

     if (rc == ERROR)
     {

          printf("Something fishy is going on around here..."\n);

          /* return rc to the operating system */

          exit(rc);

     }

     /* return 0 to the calling function */

     return 0;

}
```

In the `main()` function, the program is exited if the argument count (`argc`) is less than 3. The statement

```
exit(1);
```

tells the program to exit and return the number 1 to the operating system. The operating system can then decide what to do based on the return value of the program. For instance, many DOS batch files check the environment variable named ERRORLEVEL for the return value of executable programs.

Cross Reference:

VIII.5: Should a function contain a `return` statement if it does not return a value?