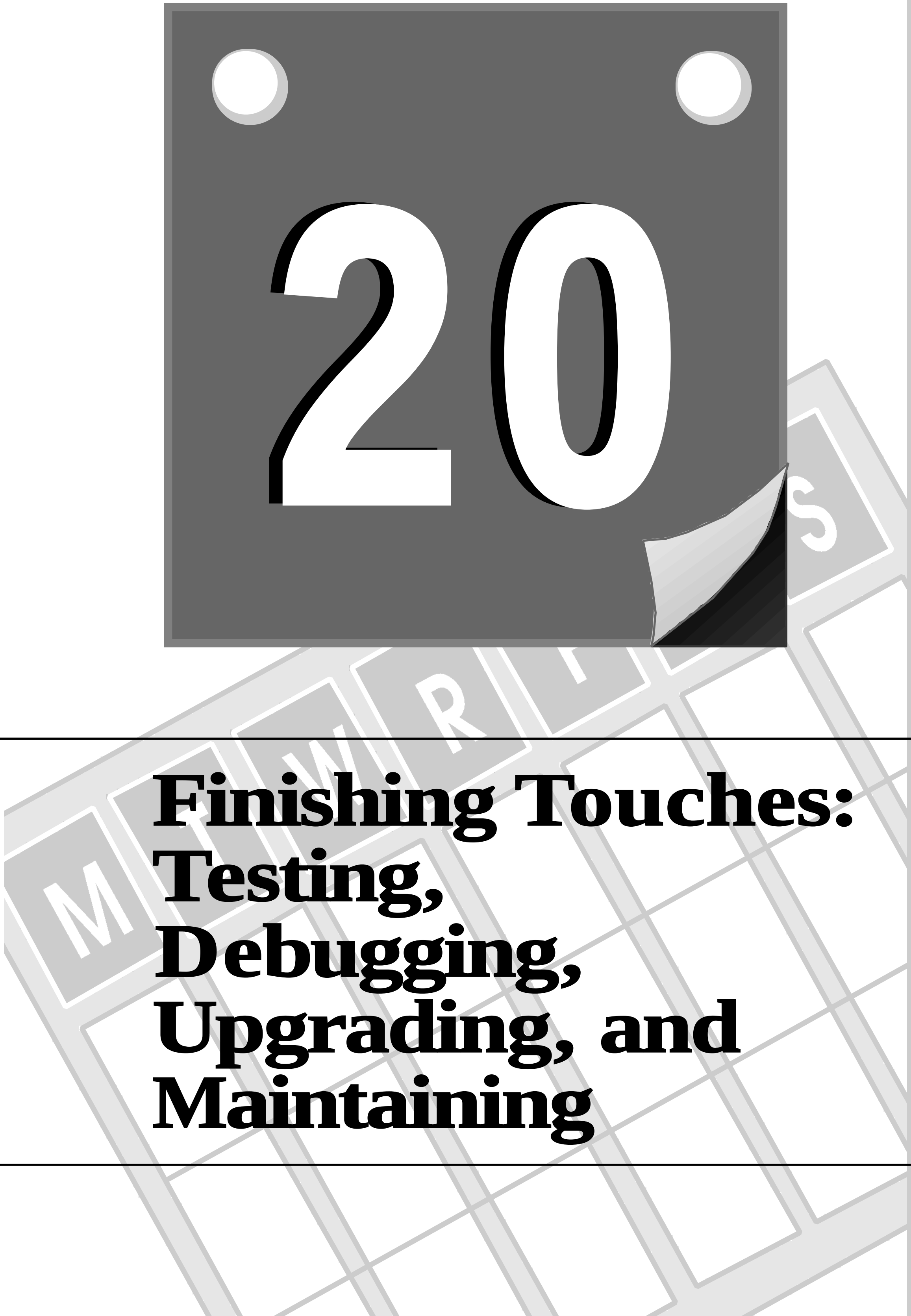

20

Finishing Touches: Testing, Debugging, Upgrading, and Maintaining

WEEK
3

Although your application is finished being coded, you shouldn't consider your job complete. There is still the need to test. In the process of testing (and in constructing) an application, you may encounter problems that need to be resolved. In addition, once others use your software, you'll begin to get feedback of suggested changes or additions. Today you will learn:

- ☐ Why testing is important.
- ☐ The different types and stages of testing.
- ☐ The different categories of problems that may be found in an application.
- ☐ Approaches to correcting common problems.
- ☐ When an application is truly complete.
- ☐ What happens after a program is released to others.

Why Test?

No program is perfectly written. In addition, when programming larger programs, some requirements may have been forgotten or worse, changed. Testing should be done to ensure the accuracy of the application and that all the requirements were included. There are two things to consider about testing. The first is what level of complexity you should incorporate into your testing. Second, you'll need to consider what types of tests will be performed.

The Complexity in Your Testing

The objective of testing is to ensure that there are no problems or errors in your program. In addition, testing should confirm that your program does what was intended. The level of detail involved in testing may determine the likelihood of problems slipping through without being caught.

> **Expert Tip:** Have someone test the application who is not familiar with it. Because you have worked so closely to the application, you'll have a tendency to overlook things. A person unfamiliar with it will lend a new perspective to it.

Many programmers consider testing to be the 15 minutes that they spend running the program right after they code it. Testing can be as informal as this; however, generally, it's better to put in a little more time and effort. Following are some of the different levels of detail that testing can entail. Generally, the greater the detail, the better the chance to find problems. The list starts with the least detailed and increases in detail.

- Free-form testing involves just using the application with no direction or guidance. Free-form testing requires no preparation. The objective is to try to cause problems just by using the software.
- The next level is testing with an objective list. To give the testing some direction, you should write down what should be accomplished by the test. For example, in the Medium Code screen, the action bar should be tested, along with each item on it, the help functions, the accelerator keys, the entry of each field, the file functions, and so on. The testing is still performed in a free-form manner; however, each objective should be covered in the free-form testing.
- Detailed checklist testing involves more work before testing begins. This is an even higher degree of testing. You write out each specific area that should be tested. For example, you can say to test the Exit on the File action bar, the Add on the Edit action bar, the Update on the Edit action bar, and so on. This ensures that every individual portion of the application is tested.
- For an even higher degree of confidence in testing, you can script the steps that the tester should follow. This is an expansion on creating the detailed checklist in that the steps the user must take to test each item are included. These steps can begin with starting the application and map each step and each key that the user should enter. In this script testing, you are ensured that everything is tested. (At least everything that is scripted.)

Note: When testing, you should enter both valid and invalid information to ensure that the application works as expected. You should ensure that the program can handle erroneous data. For example, a calculator program should be able to handle a "divide by zero" error. This is true regardless of which level of testing you perform.

Different Types of Tests

In addition to the different levels of complexity that can be applied to testing, there are also different categories of testing. There are four general categories of testing:

- ☐ Unit testing
- ☐ Integration testing
- ☐ Alpha testing
- ☐ Beta testing

> **Warning:** Many programmers don't consider testing worth the time; however, it only takes one serious problem to cause a user to not trust your program. Once the trust in your program is lost, a user will be less inclined to want to use it.

Unit Testing

Unit testing is the first category of testing. A unit is considered an individual part of an application. For example, in the *Record of Records!* application, the main menu is a separate piece of code that is compiled into an object file (.OBJ). You can test the main menu to see that it works as expected. If it does work, you don't need to recompile its source file, RECOFREC.C. If you make a change to the Medium Code screen, you'll need to recompile the MEDIUM.C source file; however, you don't need to recompile RECOFREC.C. You can still use the RECOFREC.OBJ file. Because of this, the main menu, RECOFREC.C, can be considered a separate unit from the Medium Code screen, MEDIUM.C.

> **Expert Tip:** While a unit test could span multiple source files, it's best to keep it at an object file level.

Some people choose to set up unit tests at a screen level. For an application such as the *Record of Records!* application, you could have individual tests for the main menu, the Medium Code screen, the Groups screen, and so on. Setting the unit tests at the screen level is not as good as the object level for one simple reason. If a single source file in the screen changes, then you need to re-unit test the entire screen.

A unit test can be at any of the levels mentioned in the previous section. It is recommended that you don't do free-form unit tests. At a minimum, you should have a set of objectives to test for. A unit test should be performed when all the code for the given unit is complete. There is no need to wait until the entire application is done because other parts of the application shouldn't cause changes to the unit.

If a problem is found when a unit test, or any other test, is performed, then you should fix it. If you make a change to the code, the unit test should be rerun from the beginning to ensure that nothing else changed. While this may seem tedious, it can save you a lot of embarrassment later if you find a problem.

Integration Testing

Integration testing picks up where unit testing leaves off. Unit testing tested each of the individual components of your application. Integration testing should test the integration of the units with each other. Integration testing should be devoted to testing that all the pieces work together. A large part of integration testing will revolve around navigating through different parts of the application.

The *Record of Records!* application can be used for an example of one portion of an integration test. A portion of an integration test for *Record of Records!* may start with the adding of several records, followed by searching for the records, followed by printing a report of the records. This will test to ensure that the report includes any new records that were added. Another test may involve deleting records and then trying to print the report to ensure that they were removed from the report also.

The level of complexity in integration testing is the same as the unit testing. Integration tests can be as loose knit as free-form testing; however, it is again recommended that, at a minimum, you develop a list of testing objectives. For a higher chance of removing all the errors, you can move toward integration tests that are fully scripted.

Note: An integration test should not include tests such as the following:

☐ Does the date check to see that the month is from 1 to 12?

☐ Is the dollar value greater than 0?

☐ Does the Help panel display the correct data?

These are all unit test criteria—they affect only a single unit.

Alpha Testing

Alpha testing is generally done by you or those you are working with. Alpha testing begins where unit and integration testing ends. With alpha testing, you should use the application as it is intended to be used. For an application such as *Record of Records!*, you should enter valid medium codes, musical groups, and musical items. You should also work with the reports, try all the help panels, and navigate through the action bar options. You should store your own real-live data in the application and use it as it was intended.

You may be wondering what the difference is between alpha testing and the preceding unit and integration testing. The difference is how you test. In unit and integration testing, the objective is to try to break the application so that you identify problems before you ship it out. The objective of alpha testing is the opposite. It is to use the application to ensure that it works as intended.

Beta Testing

When you first release your software, you may choose to do a beta test. Whereas you perform an alpha test, beta testing should be done by others. You should find a small number of people who will use the application. Each of these people should be given a copy to use for a predetermined period of time. As these people use the program, they should provide feedback on any problems or suggestions that they may have.

If the beta testers find any substantial changes, then you may need to restart the beta test. This would be done by updating the software and then distributing new copies of the software to each of the beta testers. You should actually release the software to the public only when the beta testers and you are comfortable with the software.

> **Expert Tip:** You should limit the number of beta testers that use your software. You should choose testers that have a high probability of using large portions of your application. The object of beta testing is to have people use actual data in your application. This provides confirmation that your program doesn't have any dormant problems.

Expert Tip: Beta testers are generally given their copies of the pre-release software. In addition, beta testers are also generally given production copies once the software is ready for public release.

Resolving Problems

How many programs have you written that have compiled and run perfectly the first time? If your answer is one or more, then you may be an exceptional programmer. Even when typing in code from a book such as this one, it's not uncommon to introduce mistakes. Before being able to fix problems, you need to be aware of the type of problems that you can encounter.

Types of Problems

A mistake or error in a computer program is called a *bug*. There are two major classifications for computer bugs:

- Syntax errors
- Logic errors

The Easy Bugs: Syntax Errors

Syntax errors are easy to spot because the compiler tells you about them. *Syntax errors* are errors in the syntax or code that cause the compiler to have problems.

Some syntax problems cause errors, others cause only warnings. If only warnings are caused, the program will still compile and create an executable program. If even one error is caused, then the compiler won't create an executable program. If you compile the following listing, you should get syntax errors similar to those presented in the output.

Type **Listing 20.1. SYNTAX.C. A program with syntax errors.**

```
/* Function: syntax.c
 *
 * Author:   Bradley L. Jones
 *
```

continues

Listing 20.1. continued

```
5:     * Purpose:  A program with syntax errors
6:     *------------------------------------------------------*/
7:
8:    #include <stdio.h>
9:
10:   int main( void )
11:   {
12:     char x;
13:
14:
15:     if( x == 1 )
16:     {
17:         printf("Hey! The variable x is equal to 1!!!");
18:     }
19:     else
20:     {
21:         printf("Uh oh! The variable x is equals %d!!!", x )
22:     }
23:
24:     printf("\n\n");
25:
26:     scanf("%d", y);
27:
28:     if( y == 1 );
29:     {
30:         printf("Wow! The variable y is equal to 1!!!");
31:     }
32:     else
33:     {
34:         printf("Bogus! The variable y is equals %d!!!", y );
35:     }
36:   }
```

Output

```
*** COMPILER STUFF ***
Error syntax.c 22: Statement missing ; in function main
Error syntax.c 26: Undefined symbol 'y' in function main
Error syntax.c 32: Misplaced else in function main
Error syntax.c 36: Compound statement missing } in function main
Warning syntax.c 36: Function should return a value in function main
*** 4 errors in Compile ***
```

Analysis

As you can see, the errors presented tell you roughly what the problems are. Your compiler may present the wording slightly differently than what is presented here; however, it should pick up the same syntax errors.

Review Tip: Fix errors before warnings. A lot of times an error can cause several warnings. Because of this, you could fix multiple problems by starting with the errors.

The Difficult Bugs: Logic Errors

Logic errors are much more difficult to spot and fix. A program can compile without any syntax errors, but still contain several logic errors. A *logic error* is an error in how something is being done. Following are several examples of logic errors:

```
for( x = 1; x < 10; x++ );
{
   printf(“%d”, x);
}
```

At first glance, you may think that this is a syntax error; however, it is really a logic error. If you didn’t spot the problem, the `for` statement has a semicolon at the end of it. This means the output of this code fragment would be only the number 10 printed once. If that was what the coder intended, then there is not an error here. If the programmer expected 1 through 9 to print, then there is a logic error.

Following is another common logic error:

```
for( x = 1; x < 10; x++ )
{
    do a bunch of stuff...
    x = 2;
    maybe do some more stuff...
}
```

Resetting a variable that is used in a loop can cause the loop to go on forever. This wouldn’t cause an syntax error; however, there is a problem.

These are simple logic errors that can occur. More complex logic errors can occur as you incorporate code that should do specific functions.

Note: A logic error is any piece of code that doesn’t perform as the program specification stated it should, or as the programmer intended.

The Oops! Factor: Recursive Errors

Although there are only two types of errors, what seems like a third type is often mentioned. This is the recursive error. A *recursive error* is an error that occurs when another error is fixed. (Oops!) You should work to avoid causing recursive errors by thinking through your changes for syntax and logic errors.

Debugging

Removing bugs in code is called debugging. There are several ways of debugging a program. Generally, the method used will be based on the type of problem. With syntax errors, the compiler informs you of where the problem is. In addition to telling you the problem, it tells you approximately what line the problem is on. By now, you should be familiar with removing these kinds of problems.

For logic errors, debugging becomes much harder. Because the compiler doesn't tell you where the error is, you must find it yourself. There are several methods that you can use to search out logic problems. The most common are the following:

☐ Performing a walk-through

☐ Using print statements

☐ Using compiler functions and preprocessor directives

☐ `assert()`

☐ `perror()`

☐ Using a debugger

To help you understand these methods, each will be covered.

Performing a Walk-Through

For programs that you are going to release either commercially or as shareware, you should always do a complete walk-through. For any other program you create, you should still do a walk-through. Hmmm, sounds like you should always do a walk-through! Better yet, you should have someone else do a walk-through of your code with you. A walk-through is the task of reading each line of code in a program. Generally, the code is read in the order that it would be executed; however, some people choose to read it starting at line one and working through to the end.

Expert Tip: When walking through the code, it is best to walk-through each line in the order that they would be executed.

A walk-through offers many benefits. The main benefit is that you'll have a better chance of understanding all of the code in your program. Secondly, you will be able to identify unused code. You can do this by marking each line that is read. Any lines that are not marked when the walk-through is complete shouldn't be needed. A third benefit is finding code that isn't efficient. If you find that the same lines of code are in several different locations, you'll be able to pull them out into a single function.

A walk-through can also be used to find logic errors. By keeping track of the variables in the code, and by determining what each line does, you can get an idea of what the program is going to do. If you have a logic error you are trying to fix, then you should be able to find it because you are, in essence, running the program.

Note: A walk-through can be automated by using a debugger. Debuggers are covered later.

Using Print Statements

Some programmers choose to use print statements to debug a program. Using print statements is probably the easiest method for debugging; however, it isn't always effective. Using print statements is straightforward. If you are testing to see that a specific function is used, then a `printf()` call at the beginning can be used to signal that you were there. A `printf()` statement can also be used to display the content of any variables.

If you decide to use print statements, then you may also want to use preprocessor directives. By including preprocessor directives, you won't need to go through and remove all the print statements when you are ready to create a final version. Consider Listing 20.2.

Listing 20.2. LIST2002.C. A simple listing.

```
/* Function: LIST2002.c
 *
 * Author:   Bradley L. Jones
```

continues

Listing 20.2. continued

```
 *
 * Purpose:  A program to print the average and total
 *           of 10 numbers.
 *------------------------------------------------------*/

#include <stdio.h>

void main( void )
{
  int nbrs[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int avg;
  int ttl;
  int ctr;

  for( ctr = 1; ctr <= 10; ctr++ )
  {
     ttl += nbrs[ctr];
  }

  avg = ttl/10;

  printf(“\nThe total of the numbers is:   %d”, ttl );
  printf(“\nThe average of the numbers is: %d”, avg );
}
```

Output

```
The total of the numbers is:   960
The average of the numbers is: 96
```

Warning: Your output for this listing and the next may be slightly different.

Analysis This listing has a problem. It prints the total and average of 10 numbers. The actual total and average should be 55 and 5.5, not what was shown in the output. This program has a problem that you may be able to easily spot; however, in a more complex program, a similar problem may go undetected. Using print statements, you can easily figure out the problem with this listing. Listing 20.3 contains this same listing with print statements included. This listing also includes preprocessor directives for removing the print statements.

Listing 20.3. LIST2003.C. The simple listing updated.

```
/* Function: LIST2003.c
 *
 * Author:   Bradley L. Jones
 *
 * Purpose:  A program to print the average and total
 *           of 10 numbers with print statements.
 *-------------------------------------------------------*/

#include <stdio.h>

void main( void )
{
  int nbrs[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int avg;
  int ttl;
  int ctr;

#ifndef NDEBUG

  printf(“Starting program....\n”);

#endif

  for( ctr = 1; ctr <= 10; ctr++ )
  {
     ttl += nbrs[ctr];

#ifndef NDEBUG

  printf(“In loop, ttl = %d, ctr = nbrs[%d] = %d.\n”,
                 ttl, ctr, ctr, nbrs[ctr]);

#endif

  }

#ifndef NDEBUG

  printf(“Done with loop\n”);

#endif

  avg = ttl/10;

  printf(“\nThe total of the numbers is:   %d”, ttl );
  printf(“\nThe average of the numbers is: %d”, avg );
}
```

Output

```
Starting program....
In loop, ttl = 988, ctr = nbrs[1] = 1.
In loop, ttl = 991, ctr = nbrs[2] = 2.
In loop, ttl = 995, ctr = nbrs[3] = 3.
In loop, ttl = 1000, ctr = nbrs[4] = 4.
In loop, ttl = 1006, ctr = nbrs[5] = 5.
In loop, ttl = 1013, ctr = nbrs[6] = 6.
In loop, ttl = 1021, ctr = nbrs[7] = 7.
In loop, ttl = 1030, ctr = nbrs[8] = 8.
In loop, ttl = 1040, ctr = nbrs[9] = 9.
In loop, ttl = 1040, ctr = nbrs[10] = 10.
Done with loop

The total of the numbers is:   1040
The average of the numbers is: 104
```

Analysis From the output, you should be able to see two different things. The first is that the value of `ttl` starts out too high. Looking at the code, you will find that it was never initialized. Secondly, you should notice that the subscripts start at 1 and go to 10. They should have started at 0 and gone to 9. Knowing this, you can fix the code and recompile a perfect program.

You should notice that preprocessor directives were used in this listing. If you recompile this listing with `NDEBUG` defined, the print statements won't be included. `NDEBUG` can be defined in several ways. The best way to define it is by including a flag when you compile. On the command line, this is done with the `/D` flag if you are using a Microsoft or Borland compiler. If you compile this with `NDEBUG` defined, the results are as follows:

Output

```
The total of the numbers is:   960
The average of the numbers is: 96
```

Using Compiler Functions and Macros

The ANSI standard defines functions or macros that can help you in finding some problems. These are `assert()` and `perror()`. Each has a different use that should be examined separately.

Using *assert()*

The `assert()` macro is used to print a message when a predefined condition occurs. If the predefined condition occurs, then the program automatically exits with the following message:

```
Assertion failed: test_condition, file filename, line line_number
```

The `test_condition` is the condition that has been set up. The `filename` is the name of the source file that the error occurred in. The `line_number` is the number of the line in the source file where the error occurred.

The `assert()` macro is used in program development to determine where logic errors are occurring. The format for using the `assert()` macro is as follows:

```
assert( condition );
```

When you use the `assert()` macro, you'll want to make sure that the condition is one that you expect to always remain true. Consider the program in Listing 20.4.

Listing 20.4. ASSERT.C. Using the `assert()` macro.

```
/* Function: Assert.c
 *
 * Author:   Bradley L. Jones
 *
 * Purpose:  Demonstrate assert()
 *-----------------------------------------------*/

#include <stdio.h>
#include <assert.h>

void print_a_string( char * );

int main( void )
{

  char *str1 = “test”;
  char *str3 = “”;
  char *str2 = NULL;

  printf( “\nTest 1:\n” );
  print_a_string( str1 );

  printf( “\nTest 2:\n” );
  print_a_string( str2 );

  printf( “\nTest 3:\n” );
  print_a_string( str3 );

  printf( “\nTest completed\n” );

  return 0;
}

```

continues

Listing 20.4. continued

```
35:  void print_a_string( char *string )
36:  {
37:    assert( string != NULL );
38:
39:    printf("The value of the string is: %s\n", string);
40:  }
```

Output

```
Test 1:
The value of the string is: test

Test 2:
Assertion failed: string != NULL, file assert.c, line 37
Abnormal program termination
```

Analysis

This program displays the `assert()` error the first time the `print_a_string()` function receives a string that is NULL. Notice that the condition in the `assert()` in line 37 has the condition of `string != NULL`. As long as this condition is true, the assertion is ignored. When it evaluates untrue, then the assertion takes over, prints the assert message, and terminates the program. As you can see, the second string that was passed to `print_a_string()` was `NULL`. The third string was never passed because the `assert()` function terminated the program.

You don't need to remove the `assert()` calls from your program when you are done with them. The `assert()` macro has been set up in a manner that enables you to turn it off by defining a constant called `NDEBUG` (no debug). If you compile Listing 20.4 with `NDEBUG` defined, the `assert()` macro is ignored. You can define `NDEBUG` in several ways. You can use the `#define` preprocessor directive at the beginning of your listing in the following format:

```
#define NDEBUG
```

This requires a coding change and, therefore, isn't the optimal solution. Another solution is to define `NDEBUG` when you compile the listing. This can be done using the `/D` in the following manner if you are using the Borland Turbo C compiler:

```
TCC /DNDEBUG ASSERT.C
```

If you are using a Borland compiler or Microsoft compiler, then the `/D` parameter is used in the same manner. If you are using a different compiler, then you should check its manuals for the proper method of defining a constant when compiling. Following is the output for the ASSERT.C listing with the `NDEBUG` constant being defined:

Output

```
Test 1:
The value of the string is: test

Test 2:
The value of the string is: (null)

Test 3:
The value of the string is:

Test completed
```

Using *perror()*

The `perror()` function can also be used to help understand problems in your programs. The `perror()` function is used to display a descriptive message for the last system error that occurred. `perror()` prints a message to the `stderr` stream. The `stderr` stream is usually the screen. The prototype for `perror()` is as follows:

```
void perror( const char *s );
```

As you can see, the `perror()` function takes a string as a parameter. When `perror()` is called, this string is used to precede the last system error that occurred.

You may be wondering what kinds of system errors would allow a program to continue executing. The ERRNO.H header file contains the numeric values and symbolic constants for many of the errors that can be set into a behind-the-scenes variable called `errno`. Most of these errors occur because of file I/O. For example, when you open a file, you may not be successful. In the event that the file does not exist, `errno` is set to `ENOENT`, which is the value 2. Consider the following listing. If the `fopen()` fails, then the `errno` variable is set.

Type

Listing 20.5. PERROR.C. Using the `perror()` function.

```
/* Function: perror.c
 *
 * Author:   Bradley L. Jones
 *
 * Purpose:  Demonstrate perror()
 *-------------------------------------------------------*/

#include <stdio.h>
#include <errno.h>     /* needed for error numbers */


int main( void )
{
  FILE *fp1;
```

continues

Listing 20.5. continued

```
  FILE *fp2;


  if( (fp1 = fopen( “perror.c”, “r”)) == NULL )
  {
      perror(“Big Problem Opening 1st File”);
  }
  else
  {
     fclose(fp1);
  }

  if( (fp2 = fopen( “abcdefg.hij”, “r”)) == NULL )
  {
      perror(“Big Problem Opening 2nd File”);
  }
  else
  {
     fclose(fp2);
  }

  return 0;
}
```

Output

```
Big Problem Opening 2nd File: No such file or directory
```

Analysis

The output displayed shows the program being used with a file that exists and a file that does not exist. As you can see, the `perror()` function concatenates the system error message to the end of the message passed in the `perror()` function call. Using the `perror()` function, you can display descriptive information to the user. You should note, however, that the `perror()` command doesn’t take any corrective action in regard to the error. This is left to you.

DO | **DON’T**

DO use the `NDEBUG` constant to remove `assert()` commands, rather than actually removing them from your listings.

DO use `perror()` to help display descriptive messages for system errors.

DON'T forget to provide corrective actions if your program has an error.

DON'T forget to include the ERRNO.H header file when using the `perror()` function.

Using a Debugger

A debugger is an automated testing tool that is provided with most of the major compilers. Many companies give their compilers names. For example, Borland's debugger is called Turbo Debugger. Microsoft's debugger is called CodeView.

While every debugger is different, most offer similar capabilities. Following is a list of several tasks that a debugger will enable you to do. By looking at the items in this list, you should be able to see why a debugger could help you find coding problems.

Most debuggers can enable you to:

☐ View your code as the program runs.

☐ View the values stored in variables at any time during the execution of a program.

☐ Run a program one line at a time.

☐ See the assembler-level code that your program is converted to when compiled.

☐ Change the value of variables while the program is running.

☐ View or change what is stored in memory.

☐ And much more.

In addition to the debugger that comes with your compiler, you can also purchase other debugging tools. In addition to stand-alone debuggers, there are also special-purpose debugging tools. Some tools work to ensure that your system is working with the operating system, others work to ensure that you are using memory properly, some show you what is happening at a machine level. Many of the supportive tools serve a purpose. Depending on the complexity of your programs, these additional tools may be worth the investment.

> **Note:** I use a program called Bounds Checker. This program is used to ensure that you are using memory appropriately. Pointers cause a C programmer more problems than anything else. Bounds Checker tracks your use of pointers and memory. Bounds checker will inform you when you use an uninitialized pointer. In addition, it will tell you when you neglect to free memory. Because memory errors can be extremely hard to track down, a tool such as Bounds Checker can help to prevent many serious problems.

When Is an Application Complete?

When you release a program to the public, you will invariably get feedback from some of the users. Some users will make suggestions for improvements for your application. Others will point out even the smallest of problems with your applications. You should welcome these suggestions and comments.

At some point, you may choose to incorporate some of the suggestions and fix the noted errors. In addition to these changes, you may also decide to make your own additions to the application. Updating software with changes and enhancements such as these is a common occurrence.

Three terms are used in making changes to your software. These are updates, patches, and upgrades. A patch and an update are almost the same thing. When your program has a serious problem that needs to be fixed as soon as possible, the change can be distributed as a patch. A patch contains only what is needed to make the fix. An update is similar to a patch in that it is a fix. An update is generally not as critical as a patch. If the use of the software is not inhibited, then an update can be put out.

An upgrade is different. An upgrade may include patches and updates; however, an upgrade will generally include updates to the software's functionality also. Upgrades should include any patches and updates that were already sent out at the initial release or last upgrade to the software. When an upgrade is created, it is often released in the same manner as the initial software. This includes adding a charge. If a charge is added, then the upgrade should add enough functionality to warrant the price of the upgrade.

Generally, with the release of patches, updates, and upgrades, comes the changing of the version number. Patches don't always warrant version number changes; however, updates and upgrades almost always do. By changing the version number, it helps you to know what changes have been made. There is a pattern to how most version numbers are changed.

Consider the *Record of Records!* application developed in this book. This application was given the version number 1.00. This is the first release of the software, hence the version number is set to 1.00. Each time an update or upgrade is sent out, a new version number should be assigned. The change to the version number is generally reflective of the level of the change. For example, if an update is being incorporated to fix a bug, then the version may be changed to 1.01 to signify that the software has been changed. When a change to the program's functionality is done in an upgrade, then you may consider renumbering the version to 1.1, 1.5, or 2.0, depending on the size of the upgrade. If you are going to change the number to 2.0, then there should be highly notable changes in the software.

Note: Some companies number their beta release of the software with a number less than one.

This section started with the question: When is an application complete? You should have the answer to this question by now. If you offer your software for sale, odds are that your application will never be complete. There will always be upgrades that you can make.

DO **DON'T**

DON'T send out an upgrade version of your software sooner than six months after the last release. It's best not to update more than once a year.

DO send out a patch if an error can cause problems for the user.

Summary

Today, you covered a lot of ground. First, you were presented with information on the importance of testing and the different types of testing. Every application should receive some testing. This testing can be unit testing, which tests only a portion of an application, integration testing, which tests the interaction of an application's parts, alpha testing, which tests to ensure that the program performs as it was intended, and beta testing, which provides feedback from actual users.

In addition to testing, debugging was also covered. Debugging is the process of removing bugs, or errors, from a program. These can be syntax errors, which are found by the compiler, or logic errors, which require more effort to remove.

Once a program has been coded and tested, it is ready to be released; however, it may not be complete. Once a program is given to users, they will generally provide feedback. From this feedback, patches, updates, or upgrades may need to be made. A patch is a single fix for a critical problem. Updates are fixes for problems or enhancements that are not critical to the application. An upgrade is a larger release of the program that may contain fixes and major updates and enhancements to the program.

Q&A

Q What is the objective of a walk-through?

A The objective of a walk-through is the same as testing—to find any problems and to ensure the software works as specified. Many people do two separate walk-throughs. The first is a code walk-through to ensure the code is accurate, efficient, and, if required, portable. The second is a logic walk-through that checks to ensure the code is doing what is expected based on the initial specification for the program.

Q Is a debugger worth the time it takes to learn?

A Yes. You don't need to know everything about a debugger to get value from it. Most people start by learning how to execute their programs a line at a time. This, along with viewing variable contents, is generally an easy process. You should learn how to use a debugger one process at a time.

Q Should additional debugging tools be purchased?

A Generally, most people don't have the need for tools beyond the debugger that comes with their compiler. If you are developing a large-scale application, then tools such as memory viewers can be worth the investment. You must weigh the cost of the tool with the expected results. For most people doing development at home, the additional cost isn't justified.

Q Do version numbers always follow a sequential numbering?

A No. Some companies choose to skip version numbers for multiple reasons. Generally, this is done when the program has versions on multiple platforms. For example, consider having a program for DOS with a version number of 3.0. If you create a first release of a Windows product, although it should be version 1.0, you may choose to give it version 3.0. If the Windows and the DOS versions provide the same functionality, then you may want the versions to be the same to avoid confusing the user.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is free-form testing?
2. What are the four categories of testing mentioned in this chapter?
3. In what order should the four categories of testing from Question 2 be done?
4. When is testing complete?
5. What is a bug?
6. What is a syntax error?
7. What are the types of errors?
8. What is debugging?
9. What is the difference between a patch and an upgrade?
10. When is a software application complete?

Exercises

1. **ON YOUR OWN:** Consult your compiler to determine what its debugger will do. Try using the debugger to walk through one of the *Record of Records!* listings.

2. **ON YOUR OWN:** Determine the type of testing that should be done with the *Record of Records!* application.
3. **ON YOUR OWN:** Test your own applications.
4. **ON YOUR OWN:** Review the software packages that you have or those in a store or magazine. Do the version numbers seem consistent?