XI

CHAPTER

Debugging

Debugging is the process of removing problems—often called bugs—from your program. A bug can be as simple as misspelling a word or omitting a semicolon, or it can be as complex as using a pointer that contains a nonexistent address. Regardless of the complexity of the problem, knowing how to debug properly can be very beneficial to all programmers.

XI.1: My program hangs when I run it. What should I do?

Answer:

There are many reasons a program might stop working when you run it. These reasons fall into four basic categories:

The program is in an infinite loop.

The program is taking longer than expected.

The program is waiting for input from some source and will not continue until that input is correctly entered.

The program was designed to delay for an unspecified period or to halt execution.

An examination of each of these situations will follow, after a discussion of the techniques for debugging programs that hang for no apparent reason.

Debugging programs that hang for no reason can be particularly difficult. You might spend hours carefully crafting a program, trying to assure yourself that all the code is exactly as intended, or you might make one tiny modification to an existing program that had previously worked perfectly. In either case, you run the program and are rewarded with an empty screen. If you had gotten erroneous results, or even partial results, you would have something to work with. A blank screen is frustrating. You don't even know what went wrong.

To begin debugging a program like this, you should look at the listing and assure yourself that each section of the program, in the order in which the sections should be executed, is working properly. For example, say that the main program consists only of calls to three functions. We'll call them functions `A()`, `B()`, and `C()`.

Start by verifying that function `A()` returns control to the main program. You can do this by placing an `exit()` command directly after the call to function `A()`, or by commenting out the calls to functions `B()` and `C()`. You then can recompile and rerun the program.

NOTE

This action could, of course, be carried out just as well with a debugger; however, this answer illustrates the classical approach to debugging. A debugger is a program that enables the programmer to observe the execution of his program, the current line it is on, the value of variables, and so forth.

What this will show is whether function `A()` ever returns control to the main program. If the program runs and exits, you will know that it is another part of the program that is hanging up. You can continue to test all the routines in this way until you discover the guilty routine. Then you can focus your attention on the offending function.

Sometimes the situation will be somewhat more complex. For example, the function in which your program hangs might be perfectly OK. The problem might be that that function is getting erroneous data from somewhere else. In this case, you will need to monitor the values your function is accepting and observe which ones are causing the undesired behavior.

TIP

Monitoring functions is an excellent use for a debugger.

An examination of a sample program will help illustrate the use of this technique:

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

/*
 *  Declare the functions that the main function is using
 */

int A() , B( int ) , C( int , int );
```

```
/*
 *  The main program
 */

int A() , B() , C(); /* These are functions in some other
                        module */

int main()
{
     int v1 , v2 , v3;

     v1 = A();
     v2 = B( v1 );
     v3 = C( v1 , v2 );
     printf( "The Result is %d.\n" , v3 );
     return(0);
}
```

After the line that invokes function `A()`, you can print the value of the variable `v1` to assure yourself that *it is within the range of values the function* `B()` *will accept*. Even if function `B()` is the one taking forever to execute, it might not be the erroneous function; rather, function `A()` might be giving `B()` values that it never expected.

Now that you've examined the method of debugging programs that simply "hang," it's time to look at some of the common errors that cause a program to hang.

Infinite Loops

When your program is in an infinite loop, it is executing a block of code an infinite number of times. This action, of course, is probably not what the programmer intended. The programmer has in some way caused the condition that keeps the program in the loop to never be false or, alternatively, has caused the condition that would make the program leave the loop to never be true. Look at a few examples of infinite loops:

```
/* initialize a double dimension array */

for( a = 0 ; a < 10 ; ++ a )
{
     for( b = 0 ; b < 10 ; ++ a )
     {
          array[ a ][ b ] = 0;
     }
}
```

The problem here is that, due to a mistake the programmer made (probably typographical in nature), the second loop, which can end only when the variable `b` is incremented to 10, never increments the variable `b`! The third part of the second `for` loop increments `a`—the wrong variable. This block of code will run forever, because `b` will always be less than 10.

How are you to catch such an error? Unless you notice that the variable `b` is never being incremented by reviewing the code, you might never catch the error. Inserting the statement

```
printf(" %d %d %d\n" , a , b , array[ a ][ b ] );
```

inside the brace of the second `for` loop is one action you might take while attempting to debug the code. You might expect the output from this code fragment to resemble this:

```
0 0 0
0 1 0
(and eventually reaching)
9 9 0
```

But what you would really see as output is this:

```
0 0 0
1 0 0
2 0 0
...
```

You would have a never-ending sequence, with the first number continually getting larger. Printing the variables in this fashion not only will help you catch this bug, but it also will let you know if the array did not contain the values expected. This error could conceivably be very difficult to detect otherwise! This technique of printing the contents of variables will be used again.

Other Causes of Infinite Loops

There are many other possible causes for infinite loops. Consider the following code fragment:

```
unsigned int nbr;

for( nbr = 10 ; nbr >= 0 ; -- nbr )
{
    /* do something */
}
```

This fragment of code will run forever, because `nbr`, as an unsigned variable, will always be greater than or equal to zero because, by definition, an unsigned variable can never be negative. When `nbr` reaches zero and is decremented, the result is undefined. In practice, it will become a very large positive number. Printing the value of the variable `nbr` inside the loop would lead you to this unusual behavior.

Yet another cause of infinite loops can be `while` loops in which the condition never becomes false. Here's an example:

```
int main()
{
    int a = 7;

    while( a < 10 )
    {
       ++a;
       a /= 2;
    }
    return( 0 );
}
```

Although the variable `a` is being incremented after every iteration of the loop, it is also being halved. With the variable being initially set to 7, it will be incremented to 8, then halved to 4. It will never climb as high as 10, and the loop will never terminate.

Taking Longer Than Expected to Execute

In some instances, you might discover that your program is not completely "locked up" but is merely taking longer than expected to execute. This situation can be frustrating, especially if you are working on a very fast computer that can perform incredibly complex tasks in minuscule amounts of time. Following are some program fragments that might take longer to execute than you would expect:

```
/*
 *  A subroutine to calculate Fibonacci numbers
 */

int fib( int i )
{
     if ( i < 3 )
          return 1;
     else
          return fib( i - 1 ) + fib( i - 2 );
}
```

A Fibonacci number is the sum of the two Fibonacci numbers that precede it, with the exception of one and two, which are set to zero. Fibonacci numbers are mathematically very interesting and have many practical applications.

NOTE

An example of a Fibonacci number can be seen in a sunflower. A sunflower has two spirals of seeds, one of 21 seeds and one of 34 seeds. These are adjacent Fibonacci numbers.

On the face of it, the preceding code fragment is a very simple expression of the definition of Fibonacci numbers. It seems, because of its minuscule length and simplicity, that it should take very little time to execute. In reality, waiting for the computer to discover the Fibonacci value for a relatively small value, such as 100, could leave one ready to collect Social Security. The following text will examine why this is so.

Say that you want to compute the Fibonacci value for the number 40. The routine sums the Fibonacci values for 39 and 38. It has to compute these values as well, so for each of these two numbers, the routine must sum two subvalues. So, for the first step, there are two subproblems, for the next step, there are four, next, eight. The result of all of this is that an exponential number of steps end up being performed. For example, in the process of computing the Fibonacci value for the number 40, the `fib()` function is called more than 200 million times! Even on a relatively fast computer, this process could take several minutes.

Another problem that might take an unexpectedly long time to solve is the sorting of numbers:

```
/*
 *  Routine to sort an array of integers.
 *  Takes two parameters:
 *    ar -- The array of numbers to be sorted, and
 *    size -- the size of the array.
 */

void sort( int ar[] , int size )
{
     int i,j;
     for( i = 0 ; i < size - 1 ; ++ i )
     {
          for( j = 0 ; j < size - 1 ; ++ j )
          {
               if ( ar[ j ] > ar[ j + 1 ] )
               {
                    int temp;
                    temp = ar[ j ];
                    ar[ j ] = ar[ j + 1 ];
                    ar[ j + 1 ] = temp;
               }
          }
     }
}
```

Upon testing this code with several short lists of numbers, you might be quite pleased; it will sort short lists of numbers very well and very quickly. But if you put it into a program and give it a very large list of numbers, the program might seem to freeze. In any case, it will take a long time to execute. Why is that?

For the answer, look at the nested `for` loops. There are two loops, one inside the other, both of them with the range of 0 to `size - 1`. This translates to the code inside of both loops being executed `size*size`, or *size squared times*! This code will perform acceptably on lists of 10 items; 10 squared is only 100. If, however, you try to sort a list of 5000 items, the code in the loop is executing 25 million times. And if you try to sort a list of one million numbers, which is not very uncommon in computer science, the code in the loop will be executed *one trillion times.*

In either of these cases, you need to be able to accurately assess how much work the code is actually doing. This assessment falls into the realm of algorithmic analysis, which is important for every programmer to know.

Waiting for Correct Input

Sometimes the program stops working because it is waiting for correct input from some source. This problem can manifest itself in several ways. The simplest way is if your program is waiting for information from the user, but you have forgotten to have it print a prompt of some sort. The program is waiting for input, but the user does not know this; the program appears to have locked up. This problem can also manifest itself in a slightly more insidious fashion due to buffering of output. This topic is discussed in more depth in FAQ XVII.1.

However, consider the following code fragment:

```
/*
 *  This program reads all the numbers from a file,
 *  sums them, and prints them.
 */

#include <stdio.h>

main()
{
     FILE *in = fopen( “numbers.dat” , “r” );
     int total = 0 , n;

     while( fscanf( in , “ %d” , &n ) != EOF )
     {
          total += n;
     }

     printf( “The total is %d\n” , total );
     fclose( in );
}
```

This program will work perfectly well, and quickly, as long as the file NUMBERS.DAT contains integer numbers—and *only* integer numbers. If the file contains anything that is not a valid integer value, the behavior of the program will be curious. When it reaches the flawed value, it will see that the value is not an integer. It will not read the value; instead it will return an error code. However, the program hasn't reached the end of file yet, so the comparison to `EOF` will not be true. Therefore, the loop executes, with some undefined value for `n`, and tries to read from the file again. And it finds the same erroneous data there. Remember, it didn't read in the data, because it was incorrect. The program will cycle endlessly, forever trying to read in the bad data. This problem could be solved by having the `while` loop also test whether correct data has been read.

Of course, there are many other possible reasons that a program might hang or otherwise appear to freeze; however, generally, the cause will be in one of these three categories.

Cross Reference:

XI.2: How can I detect memory leaks?

XI.2: How can I detect memory leaks?

Answer:

A memory leak occurs when dynamically allocated memory—that is, memory that has been allocated using a form of `malloc()` or `calloc()`—is not deleted when it is no longer needed. Not freeing memory is not an error in itself; the compiler will not complain, and your program will not crash immediately when memory is not freed. The effect is that as more and more unused memory fails to be freed, the free space available to the program for new data will shrink. Eventually, when the program tries to allocate storage, it will find that none is available. This situation can cause the program to behave oddly, especially if the programmer has not accounted for the possibility of memory allocation failing.

Memory leaks are one of the most difficult errors to detect, as well as some of the most dangerous. This is because the programming error that causes the problem can be made very early on in the development of the program, but the error will not become apparent until later, when the program mysteriously runs out of memory when it is run "for real." Looking at the line that contains the failed allocation also will not help. The line of the program that allocated memory and failed to free it might be somewhere else entirely.

Unfortunately, the C language has no built-in way to detect or fix memory leaks. Aside from commercial packages that repair or detect memory leaks, detecting and repairing memory leaks requires a great deal of patience and care on the part of the programmer. It is far better to keep the possibility of memory leaks in mind while developing the program and to exercise great caution concerning them.

The simplest, and perhaps most common, cause of memory leakage is forgetting to free memory that has been allocated for use as temporary scratch space, as in the following code fragment:

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

/*
 *  Say hello to the user, and put the user's name in UPPERCASE.
 */
```

```
void SayHi( char *name )
{
     char *UpName;
     int a;

     UpName = malloc( strlen( name ) + 1 );
                              /* Allocate space for the name */
     for( a = 0 ; a < strlen( name ) ; ++ a )
          UpName[ a ] = toupper( name[ a ] );
     UpName[ a ] = '\0';

     printf( "Hello, %s!\n" , UpName );
}

int main()
{
    SayHi( "Dave" );
    return( 0 );
}
```

Of course, the problem here is easy to see—the program allocates temporary space for the storage of the uppercase version of the name but never frees it. There is a simple way to ensure that this problem will never happen. Whenever temporary space is allocated, immediately type the corresponding `free` statement, and insert the code that uses the temporary space in between them. This method ensures that every allocated block of memory will be cleaned up when it is no longer needed, as long as the program does not somehow leave the space in between allocation and freeing by `break`, `continue`, or the evil `goto`.

If this was all there was to fixing memory leaks, it would be no problem—this is a rather trivial matter to fix. In the real world of programming, however, blocks of memory are allocated and often are needed for an undetermined period; memory leakage might result if the code that handles or deletes memory blocks is in some way flawed. For example, in the process of deleting a linked list, a last node might be missed, or a pointer that is pointing to a block of memory might be overwritten. These kinds of problems can be fixed only by careful and meticulous programming or, as has already been mentioned, by packages that track memory, or by language extensions.

Cross Reference:

XI.1: My program hangs when I run it. What should I do?

XI.3: What is the best way to debug my program?

Answer:

To know which method is best for debugging a program, you have to examine all three stages of the debugging process:

- What tools should be used to debug a program?
- What methods can be used to find bugs in a program?
- How can bugs be avoided in the first place?

What Tools Should Be Used to Debug a Program?

There are many tools that the skilled programmer can use to help him debug his program. These include an array of debuggers, "lint" programs, and, last but not least, the compiler itself.

Debuggers are really wonderful for finding logic errors in programs, and consequently they are what most programmers choose as their primary debugging tools. Debuggers commonly enable the programmer to complete the following tasks:

1. Observe the program's execution.

   This capability alone would make the typical debugger invaluable. Very frequently, even with code that you have carefully written over a period of several months, it is not always clear what the program is doing at all times. Forgotten if statements, function calls, and branches might cause blocks of code to be skipped or executed when this is not what the programmer would expect. In any case, being able to see which lines of code are being executed at all times, especially during odd behavior, gives the programmer a good idea of what the program is doing and where the error lies.

2. Set breakpoints.

   By setting a breakpoint, you can cause a program to halt its execution at a certain point. This feature is useful if you know where the error in your program is. You can set the breakpoint before the questionable code, inside the code itself, or immediately after the code. When your program encounters the breakpoint and ceases execution, you can then examine the state of all the local variables, parameters, and global data. If everything is OK, the execution of the program can be resumed, until it encounters the breakpoint again or until the conditions that are causing the problem assert themselves.

3. Set watches.

   Debuggers enable the programmer to watch a variable. "Watch a variable" means that you can constantly monitor the variable's value or contents. If you are aware that a variable should never stray out of a certain range or should always have valid contents, this capability can quickly point out the source of an error. Additionally, you can cause the debugger to watch the variable for you and halt the execution of the program when a variable strays out of a predefined range, or when a condition has been met. If you are aware of what all your variables should do, this is quite easy.

Good debuggers often have additional features that are designed to ease the task of debugging. A debugger, however, is not the only tool that can be used to debug your programs. Programs such as "lint" and your compiler itself can provide valuable insight into the workings of your code.

NOTE

Lint is a program that knows of hundreds of common programmer mistakes and points out all of them in your program. Many are not real errors, but most are worth addressing.

What these tools typically offer that a debugger cannot are *compile-time checks.* While they are compiling your code, they can look for questionable code, code that might have unintended effects, and common mistakes. Examining a few instances in which this kind of checking is used can be helpful.

Incorrect Mixing of Equality Operators

Compile-time checking can be helpful in working with the incorrect mixing of equality operators. Consider the following code fragment:

```
void foo( int a , int b )
{
     if ( a = b )
     {
         /* some code here */
     }
}
```

This kind of error can be very difficult to spot! Instead of comparing the variables, this function sets `a` to the value of `b` and executes the conditional value if `b` is nonzero! This action is probably not what the programmer intended (although it might be). Not only will the code be executed at the wrong times, but the value of `a` will be wrong when it is used later.

Uninitialized Variables

Compile-time checking can also be helpful in finding uninitialized variables. Consider the following function:

```
void average( float ar[] , int size )
{
     float total;
     int a;

     for( a = 0 ; a < size ; ++ a )
     {
          total += ar[ a ];
     }

     printf( " %f\n" , total / (float) size );
}
```

The problem here is that the variable `total` is never initialized; it therefore can, and probably will, contain some random garbage value. The sum of all the values in the array is added to this random garbage value (this part of the program is correct), and the average, plus the random garbage, is printed.

Implicit Casting of Variables

The C language will in some cases implicitly cast variables of one type into another. Sometimes this is a good thing (it saves the programmer from having to perform this task), but it can have unintended behavior. Perhaps the worst implicit cast is that of pointer-to-integer.

```
void sort( int ar[] , int size )
{
     /* code to sort goes here */
}

int main()
{
     int array[ 10 ];
     sort( 10 , array );
}
```

Again, this code is clearly not what the programmer intended. The results of actually executing this code, although undefined, will almost surely be catastrophic.

What Methods Can Be Used to Find Bugs in a Program?

The programmer should follow several tips during the debugging of his program.

Debug the Small Subroutines of Your Program; Move On to the Larger Ones Later

If your program is well written, it will have a number of small subsections. It is good to prove to yourself that these are correct. Despite the probability that the error in the program will not be in one of these subsections, debugging these subsections first will help give you a better understanding of the overall program structure, as well as verifying where the error is not. Furthermore, when examining the larger components of the program, you can be assured that this particular subcomponent is working properly.

Thoroughly Debug a Section of a Program Before Moving On to the Next One

This tip is very important. By proving to yourself that a section of code is correct, not only have you eliminated a possible area of error, but areas of the program that utilize this subsection can depend on its proper functioning. This also utilizes a good rule of thumb—namely, that the difficulty of debugging a section of code is equal to the square of its length. Thus, debugging a 20-line block of code is four times harder than debugging a 10-line block of code. It therefore aids in the debugging process to focus on one small segment of code at a time. This is only a general rule; use it with discretion and judgment.

Constantly Observe the Flow of Your Program and the Modification of Its Data

This is very important! If you have designed and written your program carefully, you should know, from watching the output, exactly which section of code is being executed and what the contents of the various variable are. Obviously, if your program is behaving incorrectly, this is not the case. There is little else to do but either use a debugger or fill your program with `print` statements and watch the flow of control and the contents of important variables.

Turn Compiler Warnings Up All the Way, and Attempt to Eliminate All Warnings

If you haven't been taking this action throughout the development of your program, this could be quite a job! Although many programmers consider eliminating compiler warnings to be a tedious hassle, it is quite valuable. Most code that compilers warn about is, at the very least, questionable. And it is usually worth the effort to turn it into "safer" constructs. Furthermore, by eliminating warnings, you might get to the point where the compiler is only emitting one warning—the error.

"Home In" on the Error

If you can go directly to the part of the program that has the error and search for it there, you can save yourself a lot of debugging time, as well as make hundreds of thousands of dollars as a professional debugger. In real life, we can't always go straight to the error. What we often do is eliminate parts of the program that could be in error and, by the process of elimination, arrive at the part of the program that must contain the error, no matter how difficult to see. Then all the debugging effort can be invested in this part of the code. Needless to say, it is very important to assure yourself that you really have eliminated the other blocks of code. Otherwise, you might be focusing your attention on a part of the program that is actually OK.

How Can Bugs Be Avoided in the First Place?

There's an old saying that an ounce of prevention is worth a pound of cure. This means that it's always better (easier) to ensure that a problem doesn't occur than to attempt to fix it after it has made its ugly presence felt. This is most certainly true in computer programming! A very good programmer might spend quite a long time carefully writing a program, more than a less experienced programmer might spend. But because of his patient and careful coding techniques, he might spend little, if any, time debugging his code. Furthermore, if at some time in the future his program is to develop some problem, or needs to be modified in some way, he most likely will be able to fix the bug or add the code quite quickly. On a poorly coded program, even a generally "correct" one, fixing a bug that has cropped up only after a period of time, or modifying a program, can be a nightmare.

Programs that are easy to debug and modify, generally speaking, follow the rules of structured programming. Take a look at some of the rules of structured programming.

Code Should Be Liberally Commented

Again, some programmers find commenting one's code to be a real drag. But even if you never intend to have someone else look at your code, it's a very good idea to liberally comment it. Even code that you have written that seems very clear to you now can become ugly and impossible to read after a few months. This is not to say that commenting can never be bad; too many comments can actually obscure the meaning of the code. But it can be a good idea to place a few lines of comment in each function and before each bit of code that is doing something important or something unclear. Here is an example of what might be considered well-commented code:

```
/*
 *    Compute an integer factorial value using recursion.
 *    Input : an integer number.
 *    Output : another integer
 *    Side effects : may blow up stack if input value is *Huge*
 */

int factorial( int number )
{
     if ( number <= 1 )
          return 1; /* The factorial of one is one; QED */
     else
          return n * factorial( n - 1 )/
     /* The magic! This is possible because the factorial of a
        number is the number itself times the factorial of the
        number minus one. Neat! */
}
```

Functions Should Be Concise

In light of the previously stated rule of thumb—that the difficulty of debugging a block of code is equivalent to the square of its length—this rule about keeping functions concise should make perfect sense. However, there's even more to it than that. If a function is concise, you should need a few moments of careful examination and a few careful tests to assure yourself that the function is bug free. After doing this, you can proceed to code the rest of the program, confident in the knowledge that one of your building blocks is OK. You should never have to look at it again. You are unlikely to have this level of confidence in a long, complex routine.

Another benefit of using small building-block functions is that after a small, functional bit of code has been defined, you might find it useful in other parts of your program as well. For example, if you were writing a financial program, you might need, in different parts of your program, to calculate interest by quarters, by months, by weeks, by days in a month, and so forth. If you were writing the program in an unstructured fashion, you might believe that you need separate code in each of these cases to compute the results. The program would become large and unreadable, in part due to repeated computations of compound interest. However, you could break off this task into a separate function like the one that follows:

```
/*
 *    Compute what the “real” rate of interest would be
 *    for a given flat interest rate, divided into N segments
 */

double Compute Interest( double Rate , int Segments )
{
     int a;
     double Result = 1.0;

     Rate /= (double) Segments;

     for( a = 0 ; a < Segments ; ++ a )
          Result *= Rate;

     return Result;
}
```

After you have written this function, you can use it anywhere you need to compute compound interest. You have not only possibly eliminated several errors in all the duplicated code, but considerably shortened and clarified the rest of the code. This technique might make other errors easier to find.

After this technique of breaking down a program into manageable components becomes a habit, you will see many subtler applications of its magic.

Program Flow Should Proceed “Straight Through”; *goto*s and Other Jumps Should Be Eliminated

This principle, although generally accepted by the computer establishment, is still hotly debated in some circles. However, it is generally agreed upon that programs with fewer statements that cause the program flow to unconditionally skip parts of the code are much easier to debug. This is because such programs are generally more straightforward and easier to understand. What many programmers do not understand is how to replace these “unstructured jumps” (which they might have learned from programming in assembly language, FORTRAN, or BASIC) with the “correct” structured constructs. Here are a few examples of how this task should be done:

```
for( a = 0 ; a < 100 ; ++ a )
{
     Func1( a );
     if ( a == 2 ) continue;
     Func2( a );
}
```

This code uses the `continue` statement to skip the rest of the loop if `a` is equal to 2. This could be recoded in the following manner:

```
for( a = 0 ; a < 100 ; ++ a )
{
     Func1( a );
     if ( a != 2 )
          Func2( a );
}
```

This code is easier to debug because you can tell what might be executed and what might not, based on the braces. How does this make your code easier to modify and debug? Suppose that you wanted to add some code that should be executed at the end of the loop every time. In the first case, if you noticed the `continue`, you would have to make complex changes to the code (try this; it's not intuitively obvious!). If you didn't notice the `continue`, you would get a hard-to-understand bug. For the second program fragment, the change would be simple. You would simply add the new function to the end of the loop.

Another possible error can arise when you are using the `break` statement. Suppose that you had written the following code:

```
for( a = 0 ; a < 100 ; ++ a )
{
     if ( Func1( a ) == 2 )
          break;
     Func2( a );
}
```

This loop proceeds from one to 100—assuming that the return value of `Func1()` is never equal to 2. If this situation ever occurs, the loop will terminate before reaching 100. If you are ever to add code to the loop, you might assume that it really does iterate from 0 to 99 based on the loop body. This assumption might cause you to make a dangerous error. Another danger could result from using the value of `a`; it's not guaranteed to be 100 after the loop.

C enables you to account for this situation, by writing the `for` loop like this:

```
for( a = 0 ; a < 100 && Func1( a ) != 2 ; ++ a )
```

This loop explicitly states to the programmer, "Iterate from 0 to 99, but halt iteration if `Func1()` ever equals 2." Because the entire exit condition is so apparent, it will be difficult to make a later mistake.

Function and Variable Names Should Be Descriptive

Creating function and variable names that are descriptive will make the purpose of your code much clearer—and can even be said to make your code self-documenting. This is best explained by a few examples.

Which is clearer:

```
y=p+i-c;
```

or

```
YearlySum = Principal + Interest - Charges;
```

Which is clearer:

```
p=*(l+o);
```

or

```
page = &List[ Offset ];
```

Cross Reference:

None.

XI.4: How can I debug a TSR program?

Answer:

A TSR (terminate and stay resident) program is one that, after executing, remains resident in the computer's memory and continues to carry out some task. It does so by making some element of the computer's operating system periodically invoke the code that the TSR program has caused to remain resident in the computer's memory.

The way that TSR programs operate makes them very hard to debug! This is because, to the debugger, the program only truly executes for a very short period. The debugger really has no way of knowing exactly what the program is doing, and it has no way of knowing that the TSR program continues to run after it appears to have terminated. The very "invisibility" that makes TSRs so useful can cause immense problems!

Furthermore, the process whereby the program makes itself resident in memory, by changing vectors, by changing the size of free memory, and by other methods, can catastrophically interfere with the execution of the debugging program. It is also possible that the debugger might clobber the changes that the TSR has made.

In any case, unless you have a debugger specifically developed for TSR programs, using a debugger probably will not be possible. There are, however, other methods of debugging TSR programs.

First, you can reuse a method described earlier, namely, that of using `print` statements to monitor the progress of a program, but with slight modifications. Whenever the TSR program is invoked by the system by whatever method is chosen (keystroke, timer interrupt, and so on), you can open a log file in append mode and print messages to it that inform the programmer about the execution of the program. This could include functions that the flow of execution encounters, the values of variables, and other information. After the TSR program is finished running (or it crashes), you can examine the log file and gain valuable insight into the problem.

Another method is to create a "dummy" TSR program. In other words, create a program that would function as a TSR, but don't make it one! Instead, make it a subroutine of a testing program. The function that would normally accept the system interrupts could easily be modified to accept function calls from the main program. The main program could contain "canned" input that it would feed to the TSR, or it could accept input dynamically from the programmer. Your code, which otherwise behaves like a TSR, never installs itself in computer memory or changes any of the operating system's vectors.

The second method has several major benefits. It enables the programmer to use his customary debugging techniques and methods, including debuggers. It also gives the programmer a better way to watch the internal operation of his program. Furthermore, real TSR programs install themselves in memory and, if they are not removed, permanently consume a section of the computer's memory. If your program is not debugged, there is, of course, a chance that it is not removing itself from computer memory properly. This would otherwise lead to complete exhaustion of computer memory (much like a memory leak).

Cross Reference:

None.

XI.5: How do you get a program to tell you when (and where) a condition fails?

Answer:

In any program, there are some conditions that should never occur. These conditions include division by zero, writing to the null pointer, and so forth. You want to be informed whenever such conditions occur in your program, and furthermore, you want know exactly where they occur.

The C language comes with such a construct, in the form of the `assert()` command. The `assert()` command tests the condition inside its parentheses, and if the condition is false, it takes these steps:

1. Prints the text of the condition that failed.
2. Prints the line number of the error.
3. Prints the source code file that contains the error.
4. Causes the program to terminate with an error condition.

Stated succinctly, the `assert()` command is intended to ensure that conditions that should never occur do not. Take a look at what a few of these conditions might be.

One of the most common problems is being unable to allocate memory. If the memory is absolutely needed and there is no way to free some, there is little choice but to leave the program. An assertion is a good way to do this:

```
foo()
{
     char *buffer;
     buffer = malloc( 10000 );
     assert( buffer != NULL );
}
```

This means that if `buffer` is ever equal to `NULL`, the program will terminate, informing the programmer of the error and the line. Otherwise, the program will continue.

Another use of `assert()` might be this:

```
float IntFrac( int Num , int Denom )
{
     assert( Denom != 0 )
     return ( ( float ) Num ) / ( ( float ) Denom );

}
```

This use prevents the program from even dividing by zero.

It should be emphasized that `assert()` should be used only when the falsity of the condition would indicate catastrophic failure; if possible, the programmer should attempt to create code to handle the error more gracefully. In the preceding example, a special error value might be assigned to fractions with a zero denominator. This does not, however, mean that the `assert()` function is useless. A well-designed program should be full of `assert`s. After all, it is better to know that a disastrous condition is occurring than to be blissfully unaware of it (or, perhaps, unhappily aware!).

Another benefit of `assert()` is that by inserting the macro `NDEBUG` (no debugging) at the top of a program, you can cause all the asserts to be ignored during the compile. This is important for production versions of a program, after all the bugs have been fixed. You can distribute a version without the debugging code in the binary but, by removing the definition of `NDEBUG`, keep it in your version for debugging value. The code without all the tedious checks runs faster, and there is no chance of a customer's program suddenly stopping because a variable has strayed slightly out of range.

Cross Reference:

XI.1: My program hangs when I run it. What should I do?

XI.3: What is the best way to debug my program?