Techniques of Debugging

by John Locke
(jxxl@taurus.cs.nps.navy.mil)
Computer Science Department, Naval Postgraduate School

1. Introduction

Debugging is commonly thought of as a process that occurs after a program fails. That's certainly true. But since many techniques both diagnose and prevent problems, debugging can also be considered "preventative medicine," discipline that keeps programs from failing in the first place. If there's a single, methodical approach to debugging, it's the structured design that's done *before* any code is written; in the case of serious program failures, stronger design may have to be applied retroactively to straighten the mess out, but this is by far the more painful approach. A thorough discussion of structured programming would push debugging into the realm of software engineering, which is beyond the scope of this paper. Here, instead, we examine a few specific methods for avoiding and diagnosing common, small-scale problems.

The examples use C and UNIX but can easily be extended to C++. Applicability to other languages and systems may vary.

2. Getting Information out of a Program

A program can be thought of as an arrangement of data--data structures--and the operations that manipulate the data. The operations are easiest to understand; they are the program code that's laid out on the page. Data, however, exists in memory where it is invisible unless we take measures to examine it. It is therefore of paramount importance that we keep track of the data, its structure and contents.

2.1 Print Statements

Printing the contents of data elements is a fundamental programming technique, particularly useful for simple variables. For example:

```
for (i = 0; i < 10; i++) {
        /* do something */
        printf("i = %d\n", i);
}
```

Each pass through the *for* loop is printed with the loop number. In conjunction with other print statements, this gives a good picture of what the loop accomplishes.

With any reasonably complicated program, a great quantity of print information can be quickly generated. It's therefore useful, as a debugging technique, to slow down the program and observe its operation in "real-time." This can be easily done with the *sleep()* call which causes the executing process to be taken off the operating system (OS) run queue for a specified number of seconds:

```
for (i = 0; i < 10; i++) {
        /* do something */
        printf("i = %d\n", i);
        sleep(2);
}
```

In this example, the process stops executing for two seconds per loop iteration, plenty of time to observe the process in action and read the print statements as they scroll.

An occasional problem results because *printf()* output is buffered, that is, for efficiency the OS accumulates a number of bytes of output before actually printing them to the screen. If the process crashes on code after the message is buffered but before it prints, you will never see it. The solution is to flush the buffer immediately with *fflush()*:

```
for (i = 0; i < 10; i++) {
        /* do something */
        printf("i = %d\n", i);
        fflush(stdout);
}
```

2.2 Preprocessor Directives

When a bug has been corrected and the program runs correctly, print statements may no longer be desired. Our first impulse is to delete them from the code, cleaning up the source and making the program efficient. Inevitably, though, old bugs come out of hiding and new bugs are born, necessitating the work of adding the print statements back again. The solution is to segregate debugging code from the final working code through the use of preprocessor directives. These directives make it easy to choose whether to include chunks of code into the compiled program. Consider this example:

```
#define DEBUG
...
#ifdef DEBUG
        printf("Reached checkpoint 1\n");
        fflush(stdout);
#endif
```

The line *#define DEBUG* sets the user-selected label *DEBUG* equal to the Boolean value "true." *#ifdef DEBUG* tests the value of *DEBUG*, and if true compiles the following code up to the *#endif*. There can be many separate sections of code bracketed with *#ifdef DEBUG* and *#endif*. The code they bracket can all be compiled with *#define DEBUG*; it can all be excluded by either deleting the *#define*, commenting it out, or replacing the *#define* with *#undef DEBUG*. Additionally, there can be multiple labels defined, e.g. *#define DEBUG*, *#define DEBUG2*. As a result, we have an easy method of excluding debugging code from the compiled program without actually deleting it from the source.

(For ways of defining preprocessor labels without editing the source, refer to the companion document *Using Make to Maintain Software*.)

There are a couple of pitfalls to avoid when using preprocessor directives. Consider this example:

```
for (i = 0; i < 13; i++)
        j = i * 5;
#ifdef DEBUG
        printf("i = %d, j = %d\n", i, j);
#endif
```

The *for* loop works fine whether or not *DEBUG* is defined. Implicitly, however, we intend the debugging statement to print at every iteration. But it will only print after the last loop iteration completes, though, because the *for*, like many C constructs, only repeats the single following line unless brackets are used to group subsequent lines. This is how the code should be written to work whether or not *DEBUG* is defined:

```
for (i = 0; i < 13; i++) {
        j = i * 5;
#ifdef DEBUG
        printf("i = %d, j = %d\n", i, j);
#endif
}
```

Another preprocessor problem, which can be quite tricky to solve, is that a program runs correctly when the debugging code is included, but fails when it is not. The reason might be because we've included more than print statements between the preprocessor directives and simply overlooked the fact. It might be because the time expended by screen printing I/O allows other time-sensitive operations to succeed--this is the worst case but occurs only rarely. It could also be that in a multiple source file program, some files have been compiled with debugging code turned on and others with it turned off. A complete recompile of the program will usually fix this. For another method of addressing this problem, consult the upcoming section, *Pure C*.

2.3 Multi-level Debugging Messages

The problem with print statements is that there ends up being too many of them. It's simple to print out a huge number of messages by including the *printf()* in a loop. Often, the answers we're looking for get lost in the mess. Here we describe a simple solution.

The first step is to classify messages by their importance. For the sake of argument, we define three levels of message: 1) merely informative, 2) problem warning, and 3) critical failure. The message level and its number go together. Consider these code fragments:

```
int debug_level = 2;   /* globally defined */

/* inside a function... */

   for (i = 0; i < 1000; i++) {
      /* do something */
      if (debug_level <= 1)
         printf("i = %d\n", i);
   }
   ...
   if (file_found == FALSE) {
      if (debug_level <= 3)
         printf("Exiting, file not found!\n", i);
      exit(0);
   }
```

We have defined a global variable *debug_level*. We set this variable to the lowest level of message we wish to see printed. In this case, setting *debug_level* to 2 causes messages of level 2 and 3 to print out. In the body of the program, every message is subject to the test,

```
if (debug_level <= message_level#) printf("message");
```

Message levels can be easily controlled simply by changing the value of *debug_level.* By changing *debug_level* from 1 to 2, say, all the loop counters could be eliminated from the print.

(Note that the debugging level code can, and probably should, also be bracketed with preprocessor directives to easily eliminate it from the final version of the binary.)

2.4 System Logger

Most programs written for class assignments will run as a single process. However, thesis projects may be sufficiently advanced to have a multi-threaded architecture, two or more concurrently executing processes. Usually, such an application will be started as a single process which, in due course, either forks separate processes or defines an interrupt handler which runs outside the sequential flow of the main program. With multi-threaded programs, extracting information is more of a problem. Printed information from all the processes will mix together on the screen. Or the screen is being capitalized by a graphical display. Or the program has been started independent of any terminal, e.g. from a remote machine. In any case, the cause and effect between separately executing processes can become obscured.

The UNIX system logger provides a solution. This is a facility that allows messages to be ranked by severity, logs messages based on severity level, and sends all messages to a specified file. It also logs the ID of the reporting process and time-stamps messages by the system clock. The result is a file of all application messages, listed in precise chronological sequence.

The catch is that the system logger is an important OS service and, as such, its configuration file, where facilities and error levels are specified, can only be edited from the root account. In the case of a special need for this facility, contact the author for further details and assistance.

3. Error Checking

A general approach to error checking falls within the realm of program design and won't be discussed in detail here. Suffice it to say, in a rigorously implemented program, ten percent of the code performs the basic functionality of the program, the other ninety handles the immeasurable number of things that can go wrong. A conservative approach to programming is to code error-handling routines for every conceivable failure, even when the likelihood of such code ever being reached is small. But it's like insurance. The more you buy, the better protected you are when trouble happens; but it's more you have spent if trouble never occurs.

3.1 System and Library Calls

Often, we access OS resources in our programs, the system and library calls. Information regarding the failure of these resources is readily available from the OS. If we neglect to monitor it, debugging is complicated since this information is difficult to obtain any other way.

First of all, what are these resources? System calls are the basic OS resources for manipulating files (and hardware devices, which UNIX treats as files). Commonly used calls are *open(), read(), write(), close()*, in other words, the operations we associate with files. System calls and their proper usage are documented in Section 2 of the standard UNIX manual, and are essentially similar on different flavors of UNIX.

Library calls are the routines in the standard C library which are automatically linked to your program when you compile, or the routines in other system libraries you may use, like the math library. Library calls can be thought of as packages of code that make system calls for you while adding higher-level functionality. Examples include the buffered I/O file handling routines like *fopen(), fread(), fwrite(), fclose().* Library calls are documented in Section 3 of the manual.

We mention the manual because strict adherence to its guidelines will prevent most problems from ever occurring. As an example, let's look at the use of *fopen(),* to open a file. The first step is to check the man page with *man fopen* (some words appear in different sections; the ambiguity can be resolved by referencing the section number, e.g. *man 3 chmod*). The first important part is the synopsis:

```
SYNOPSIS
     #include <stdio.h>

     FILE *fopen(filename, type)
     char *filename, *type;
```

This tells us we must include *stdio.h* in our program, particularly if we want to use constants defined elsewhere in the man page. More important, the prototype for the call shows the precise number of arguments to be passed, their data types, and the return value. If close attention is paid to this part, many problems will never arise. Don't rely on the compiler to catch type mismatches; there are cases where it won't.

The text of the man page provides a good description of the call's purpose and the nature of the data you pass it, so obviously that should be read carefully.

The last critical section describes what the call returns:

```
RETURN VALUES
     On success, fopen() returns a pointer to FILE which
     identifies the opened stream. On failure, it returns
     NULL.
```

The "on success" part is usually expected; it's the "on failure" part that deserves close attention. In the next section, a general approach for handling these failures is described.

3.2 System Errors

UNIX puts the reason for system call failures into the global variable *errno*, and the routine *perror()* can be used to print out the reason after *errno* has been set. Here are the code fragments--continuing with the *fopen()* example--that comprise a complete use of the facility:

```
/* at the top of the source file, not in any function */
#include <errno.h>
extern int errno;

   /* inside a function */
   FILE *fp;

   if ((fp = fopen("myfile", "r")) == NULL) {
      perror("fopen() of myfile");
```

```
        exit(0);
    }
```

The important thing to note here is that the return value from *fopen()* is tested for the failure condition described in the man page. *fopen()* passes its return value to *fp*, then that value is compared to *NULL*. In a successful call, *fp* will contain a positive integer, the test will fail, and the program will proceed. On failure, *fp* will be set to *NULL*, *errno* will get a number indicating the type of failure, and the error-handling code will be executed. *perror()* will print out the string we pass as an argument as well as a text description of the error. Here's what the output of a failure might look like:

```
% myprog
fopen() of myfile: File not found
```

The message makes the problem clear. If the *fopen()* return value was *not* checked, the program would simply proceed without complaint and probably fail ungracefully later when a read from the unopened file was attempted.

Note that the coded response to the failure is *exit(0)*. This causes the process to simply terminate, an appropriate response if there is no reason for the program to proceed after failure. Most applications will require a more sophisticated approach.

4. Data Types and Sizes

Another source of problems is the inadvertent mixture of data types. This is a particular problem with C because of its inherent flexibility in manipulating data. As noted above, for system and library calls, always pay very close attention to the types of the arguments specified in the man page. You would think that the compiler would flag any failure to do so, but this is not always the case.

Consider the routine *scanf()*, used for reading input. According to the man page, this is the prototype:

```
int scanf(format [ , pointer... ] )
char *format;
```

scanf() takes a format string and a variable number of pointers to data items that will be filled by *scanf()* in accordance with the format. The common mistake is to pass the value of a data item instead of its address, e.g. *myvar* instead of *&myvar*. This problem is tricky to diagnose because the compiler can't tell whether the argument is a pointer or a variable containing a pointer, and will not report a type mismatch at compilation. In all likelihood, passing the bad argument will cause a segmentation violation (described in an upcoming section).

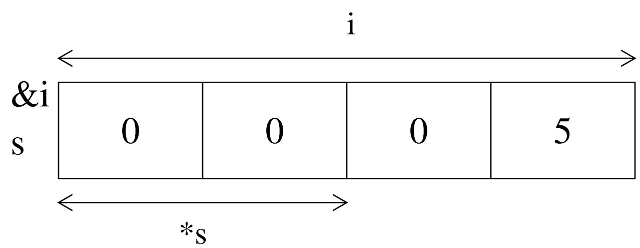
Passing the wrong type can also be a problem with user-defined functions. Consider this program:

```
main()
{
   int i = 5;

   myfunc(&i);
}

myfunc(s)
   short *s;
{
   printf("%d", *s);
}
```

What will the output of the program be? The obvious coding error is that *myfunc()* expects the address of a *short*, but we pass it the address of an *int*. Sadly, the C compiler builds the executable without complaint. Here is what's going on in memory:



&i and *s* are both pointers to the same address. But *&i* is a pointer to *int*, while *s* is a pointer to a *short*. So printing the value *s* points to, **s* in the *printf()*, outputs *0*, the contents of the two high-order bytes of *i*.

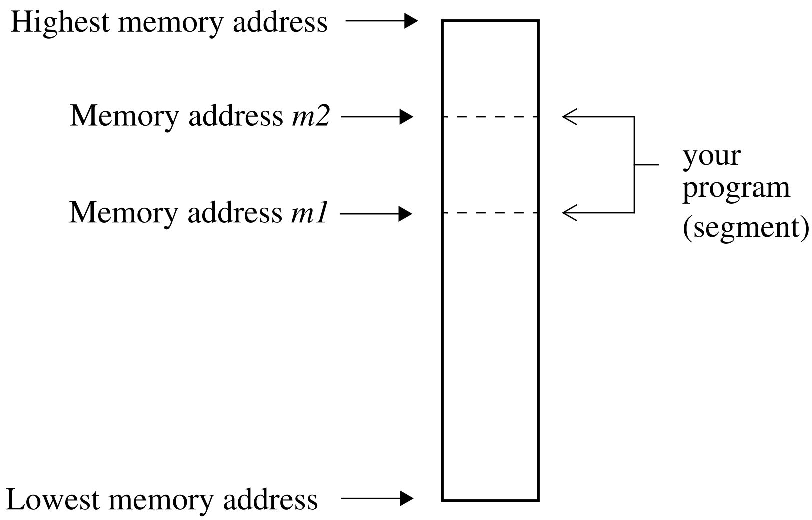
In this brief example, the problem is immediately apparent. But in a large program, this sort of mistake conceals itself well and can be time-consuming to track down. Not only does the compiler pass the error through, but the program can run without failure. The only evidence of the bug will be incorrect results produced by the program. The lesson is to know the sizes of the data types you use, and to always pay close attention to the types passing between functions.

5. Segmentation Violations

A common experience for UNIX programmers is to run a program with this mysterious result:

```
% myprog
Segmentation fault (core dumped)
```

What's a segmentation fault? This is your program running in the system's memory:



Your program occupies a specific *segment* of memory, from lower memory address *m1* up to *m2*. Because UNIX supports multi-processing, it would be disastrous if your program were allowed to write into memory addresses below *m1* or above *m2*. You would fatally disrupt the integrity of another program by accidentally writing into its segment. Worse still, if you wrote into the OS's space, you could crash the entire system. Therefore, all pointers used by your program must have values between *m1* and *m2*. Consider this simple program which causes a segmentation fault:

```
int  *ptr;

*ptr = 1;
```

ptr is declared a pointer to *int*. **ptr*, in the second line, *dereferences* the pointer, it specifies the target of the assignment. But *ptr* has never been initialized; we haven't allocated memory for it to reference. Therefore, the variable *ptr* contains either zero or whatever random bit pattern that happens to be on the stack where *ptr* resides. In any event, it's unlikely to contain a value between *m1* and *m2*. Thus, the segmentation fault.

Segmentation faults could arise from other programming mistakes--bad pointer arithmetic, for instance--but they are usually caused by dereferencing an uninitialized or a *NULL* pointer.

6. Bus Errors

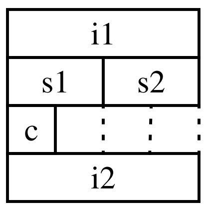
Another mysterious error message is:

```
% myprog
Bus error (core dumped)
```

Before explaining what this means, we need to look at how UNIX arranges variables in memory. UNIX generally runs on machines with hardware designed for 32-bit memory addresses, i.e. the address bus is 32 bits wide. Pointers are therefore 32-bits and fetches from memory are done in whole machine words, also 32-bits. This creates an efficiency which compilers respect by putting data elements between word boundaries and not crossing them. Consider this simple data structure:

```
struct mystruct {
   int   i1;
   short s1;
   short s2;
   char  c;
   int   i2;
};
```

The compiler stores it into memory thusly:



int i1 starts on a 32-bit word boundary, as does *short s1*. *short s2* can be packed after *s1* without crossing the word boundary. *char c* starts on a word boundary, however the three remaining bytes of the word are too few for the four-byte *int i2*, so *i2* starts on the next boundary. Even though there are only 13 bytes of data defined in the *struct,* it actually takes 16 bytes of memory. This can be verified with *sizeof(struct mystruct)*, which yields a result of 16.

Since fundamental C data elements cannot cross word boundaries, this suggests something about the validity of pointers. Pointers to *int* can only refer to word boundaries; pointers to *short* can refer to word boundaries, or word boundaries plus two; while pointers to *char* can start at the address of any byte. Consequently, any pointer not meeting the criterion for its type is invalid. Any attempt to dereference an invalid pointer will cause a bus error and program failure.

Here's a sample program that creates a bus error:

```
struct {
   short s1, s2;
} mystruct;

int *intptr = (int *) &(mystruct.s2);

*intptr = 13;
```

In *mystruct*, *s1* and *s2* are packed into four bytes with *s1* at a word boundary and *s2* at the word boundary plus two. We assign the address of the *short s2* into a pointer to *int* whose integer contents would cross a word boundary. We type cast the address to avoid a compiler warning. Dereferencing this invalid pointer to *int* causes the bus error.

In debugging a bus error, we typically look for unorthodox combinations of arrays, structures, and type casts as the likely culprits. In complicated programs, we can't rely on the compiler to flag the problem, but if the compiler does produce the warning,

```
line n: warning: illegal pointer combination,
```

it would be worthwhile to investigate.

7. The *dbx* Debugger

A detailed description of debuggers is beyond the scope of this paper. Debuggers are sufficiently well-featured that their own documentation should be consulted. However, the *dbx* debugger is generally available on UNIX systems and can be used to quickly pinpoint the source line causing a program failure, particularly segmentation faults or bus errors, and thus merits a brief discussion.

dbx is a symbolic debugger, meaning it refers to memory locations using the symbolic labels--variable, data structure, and function names--in the program's source code. Normally, these text symbols aren't included in the executable binary since they take up space both in memory and on disk--large programs can more than double in size with the inclusion of its symbols. Consequently, if *dbx* is to be used effectively, the symbol table must be explicitly included by using the *-g* flag at compile time, e.g.:

```
cc -g -o myprogram myprogram.c
```

dbx can now run the program, allowing break points to be set, allowing the program to be executed one line at a time, and allowing the content of variables to be checked at any stage in the process. However, we will simply describe using *dbx* to identify a fatal bug.

Consider the short program we used previously to force a bus error. We put the code in a *main()* routine and compile it thusly:

```
cc -g -o busprog busprog.c
```

As expected, program execution fails, reporting *Bus error (core dumped)*. The *core* is a file containing the program's memory segment, including machine code, data areas, and symbol table. The *core* is dumped (written) whenever a program fails with a bus error or segmentation violation. *dbx* needs the core to match up the machine code with the source listing. The following demonstrates a *dbx* invocation that identifies and lists the error (user input is in **boldface**):

```
% dbx busprog
Reading symbolic information...
Read 36 symbols
program terminated by signal BUS (alignment error)
```

```
(dbx) where
main(), line 9 in "busprog.c"
(dbx) list
    9      *intptr = 13;
(dbx) quit
```

dbx has its own prompt, *(dbx)*, where commands are typed. Type *help* in *dbx* for a listing of the commands. The command *where* identifies the source line and file that caused the failure (multiple source programs can be used). The command *list* displays the source line. Variable contents can be displayed with *print varname*.

8. Memory Types and Initialization

8.1 Memory Types in a Program

There are three basic types of memory in a program:

1. The global (or static) data area. Data elements declared outside of any function go in this area and remain active for the life of the program. They are automatically initialized to all zeros, but this could be compiler-dependent, so it's a good idea to initialize these data elements anyway.
2. The stack. The stack contains data elements local to functions. These elements are not initialized automatically and are very likely to contain non-zero values so it's a good practice to initialize them in the program.
3. The heap. The heap is the pool of memory where dynamically allocated data elements reside. *malloc()* and *free()* are used to obtain and release memory from the heap. Data elements in the heap are not initialized and should be done so explicitly in the program.

8.2 Initializing Data Elements

Many problems can result from uninitialized data elements. If a data element is declared on the stack, the stack pointer simply moves down the stack to make room for it. Whatever random bit pattern happened to be on the stack becomes the default value of the data element. But a common coding technique is to use a test for a non-zero (or non-NULL) value to determine the validity of a data element. Many data elements on the stack with the random default value pass this test, thus undermining the integrity of the program. Furthermore, the effect can vary with separate runs of the program, since on a multi-tasking machine the program won't always be loaded into the same block of memory. Likewise, these same problems can occur with memory obtained from the heap.

Following are some simple techniques to help avoid these problems.

- Initialize all variables. i.e. set them to their initial value:

```
int   myint = 0;
int   is_true = TRUE;
int   *intptr = NULL;
```

- Use *bzero()* to initialize more complex data elements to all zeros:

```
struct mystruct {
   int   i;
   short s;
};
struct mystruct array_of_struct[100];

bzero((void *) array_of_struct, sizeof(array_of_struct));
```

- Write standardized routines for dynamically allocating and initializing data structures:

```
struct mystruct {
   int   i;
   char  *string;
};

main()
{
   struct mystruct *local_ptr;

   local_ptr = malloc_mystruct();
}

struct mystruct *malloc_mystruct()
{
   struct mystruct *ptr;

   if ((ptr = (struct mystruct *)
                  malloc(sizeof(struct mystruct))) == NULL) {
      printf("malloc of struct mystruct failed.\n");
      return (NULL);
   }
   bzero((void *) ptr, sizeof(struct mystruct));

   return (ptr);
}
```

Note that in *main()*, the top level of the program, the code is simple and easy to read. The messiness of the proper use of *malloc()*, with error checking for failure and initialization, is buried in *malloc_mystruct()* and never need be repeated elsewhere.

The *bzero()* is an efficient way to set all variables to zero and pointers to *NULL*. However, it could be easily replaced with a set of initializations more appropriate to the needs of the program.

Additionally, *malloc_mystruct()* can be very quickly copied and edited for other data structures. A standard function naming convention like *malloc_structname()* also makes it easy to remember the name of the allocation routine.

8.3 Data Structures and Pointers

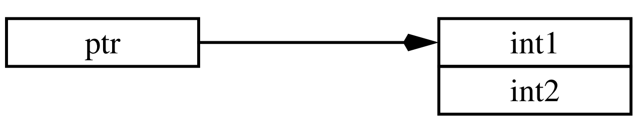
It's useful, in general, to understand how C uses memory. There are also some useful bug-preventing techniques that can be applied. Let's look first at data structures and pointers and the memory they use. Consider these variations on a theme:

	C Construct	Memory Used
1)	`struct mystruct { int int1, int2; };`	None!
2)	`struct mystruct { int int1, int2; } named_struct;`	int1 int2
3)	`struct mystruct *ptr;`	ptr

Example 1) uses no memory; it's simply a definition, a template for what the structure *would* contain if one were declared.

Example 2) both defines and declares a structure. Memory for the structure exists and can be accessed with *named_struct.int1* and *named_struct.int2*.

Example 3) allocates memory, but only four bytes, the size of a pointer. However, you cannot access *int1* and *int2* without first allocating the memory for *ptr* to point to. Attempts to utilize *ptr.int1* or *ptr.int2* cause a syntax error at compile-time because the dot operator implies the structure has been defined and has a name, as in 2). Attempts to use *ptr->int1* or *ptr->int2* are syntactically correct--the arrow symbol dereferences a pointer--but this program will fail with a segmentation violation unless A) memory for the size of *struct mystruct* is dynamically allocated with *malloc(),* or B) the address of an existing piece of memory is assigned to *ptr*, e.g. *ptr = &named_struct*. In any case, for a pointer to be useful, there must be a pairing of pointer and object pointed to, e.g.:



Carefully consider the distinctions made between the above three examples. Many program errors and *man-debugging-hours* are caused by a failure to understand the principles.

9. Pure C

Strictly speaking, not everything we include in C programs satisfies the formal definition of the C language. Our programs include both C language constructs *and* instructions to the C preprocessor. We recognize these preprocessor directives as starting with a pound sign. For instance, *#define* allows a descriptive text string to substitute for a value, making the code more comprehensible wherever the value is used. Before compilation, the preprocessor substitutes the value for the string wherever it occurs. This also gives a way of setting constant values at one place in the program even though they may be used repeatedly.

Another use of preprocessor directives is conditional compilation. This purpose is served by *#ifdef VARNAME* and *#endif*. This allows one program to be written to run on different architectures, or to include optional features, without the need to maintain separate programs.

Preprocessor directives are an important convenience, but they sometimes obscure the underlying program. It might require finding and searching header files to determine whether a preprocessor variable is defined. That variable may be nested in a conditional, forcing yet another search. Also, conditional compilation directives can be nested to many layers, making it even harder to tell what a compile will actually create.

Sometimes, rather than laboriously reading code, or if it is unclear what effect a preprocessor directive like a macro will have, it is easier to just examine the *pure C* code that has been run through the preprocessor and see what the C compiler actually compiles. Preprocessed code can be created with the compiler's *-E* flag:

```
cc -E myprog.c > myprog.cpp
```

Note that the output of *cc -E* is redirected to another file so that it can be examined in a text editor. The preprocessed code will have a lot of blank lines, so outputting it to the screen will not usually be helpful.

10. Naming Programs and Variables

Programmers are instructed to use nice descriptive names for programs and variables, as opposed to terse algebraic names or cryptic abbreviations. This is good advice but it occasionally creates mysterious problems when the nice name corresponds to the name of another program on the system, or to a keyword in the programming language.

Here's an example of keyword conflict. Say you wanted to declare an integer to hold the return value from a function, so you added this line to your program:

```
int return;
```

This is the compiler's response:

```
line n: syntax error at or near word "return"
line n: unknown size
```

Although the compiler balks, it's messages don't really specify the problem. The first message draws your attention to the sequence of symbols; the second to the *int* keyword. The real problem, of course, is that *return* is itself a keyword and can't be used as a variable name. This subtle type of problem occurs most often to programmers either starting out in a language or switching frequently between different languages.

Program naming problems come from another source, the path variable in your shell. Consider the following example. You've written a short program to test something, so naturally you name it *test*. You run *test* and though no error messages are reported, nothing seems to happen. You put in some *printf()* statements, but nothing prints out. The problem is that there is a system command called *test* used by shell scripts--run from the command line, it simply returns. When you execute *test*, the system command runs rather than your new program because the directory the system *test* resides in comes first in your path listing. Unlike MS-DOS, the current directory--the dot--does not automatically come first. It can be listed first, but these days it is usually listed last for security reasons[1]. Since it's difficult to be familiar with the hundreds of command names on a UNIX system, the command *which* provides a simple diagnosis of the problem by searching the path for the first name match:

```
% which test
/usr/bin/test
```

A sure-fire remedy is to specify a more complete path for your program:

```
% ./test
```

This guarantees the *test* that is executed is in the current directory.

The lesson here is that names should be nice but not too nice. If in doubt, make the name more elaborate, e.g. *int return_val*.

11. Rules of Thumb

In closing, here are a few more rules of thumb that will aid in (near) error-free programming:

Code in small chunks! The more code you write in between compiles, the more bugs will be generated. So proceed in small, logical steps.

Change one thing at a time! If you add two new things to a program and the program fails, you will have to take one of the new things out of the program to isolate the problem. So you might as well add one thing at a time.

1. If the current directory lists first, trojan horses surreptitiously put in your home directory, with common system names like *cc* or *ls*, will execute without your knowledge.

Initialize variables! It's better to initialize too much rather than too little. The computational expense of initialization is small, but the frustration and time expense of one bug can be extremely high.

Don't blame the system! Bugs can have extremely strange effects and sometimes misleading error messages will be produced. You might be tempted to look outside your program for answers. It's true that compilers, operating systems and other supporting software can have bugs. It's also true that hardware fails on occasion. But your program is failing because it's not written correctly. That's practically always true.

Don't give up! Sometimes the simplest of programming mistakes take hours to track down. It goes with the territory. No programmer avoids the ordeal of debugging. But a good programmer minimizes bugs by making "defensive coding" a habit.