C of Peril

Paul L Daniels

29th November 2004

Contents

Chapter 1

Preface

1.1 Introduction

C is frequently touted as being one of the most perilous languages next to assembly code (or binary machine code for that matter). A lot of troubles arise from the misconstrued public perception of many standard functions. Additionally, blatent abuse or laziness consequently contributes to many of C's perceived failures.

The best explanation for the existance of this book is that it aims to teach better C programming and demonstrate how.

1.2 Intended Audience

This book is intended to be read by C programmers of all levels. I strongly believe that getting new comers to C informed about the potential pitfalls that befront them will stand them in good stead for the years to come, hopefully eliminating many of the frustrating hours (days) involved in debugging problematic programs as often involved with the problems covered in this book.

For the advanced C programmer, I still recommend looking into this book, if only to reaffirm your understanding of what certain C functions do (and don't) do.

If it's any consulation, I was prompted into writing this book whence discovering after over a decade of programming, I had failed to realise that *strncpy()* could in fact still induce a buffer overflow situation. If nothing else I was lulled into the perception that it was safe by the masses of people touting "Replace strcpy with strncpy".

1.3 Thanks and Accreditations

At the risk of being stereotypical, I would like to thank my beautiful wife, Elita, for her endurance in a time where there simply wasn't enough time for everything that needed it.

The people of the #c IRC chat channel on the FreeNode network (formerly OpenProjects), for their endless submissions of really bad C code.

Part I

C of Peril

If you ask a veteran C programmer for a list of functions which are known problem areas, chances are they'll rattle off names such as strtok, strcpy and gets. However, knowing to avoid those functions is only part of the way towards resolving the issue, instead, it would be nice to know *why* those functions (and others) are problematic.

Most programming problems related to C functions tend to be due to gross public misconceptions about how the functions work, in so far as having witnessed several tutoring sessions and books which incorrectly depict matters.

Chapter 2

Character / String Functions

One thing you will note about a lot of C's "bad" functions, is that they're typically associated with strings. C does not actually have a string type, instead it defines a string as a sequence of characters, terminated by a NUL (\0) character. The NUL character is of utmost importance, without its presence, string processing functions can continue on far beyond the actual allocated space (for the variable in question) and produce some rather interesting results.

2.1 strcpy

In short, you should never use this function. It ranks possibly as the Number 1 of all causes of buffer overflows in programs written in C. The problem with strcpy is that you [the programmer] never quite usually know how big the source is going to be, and worse still, you have no way of defining how big the destination is. A very simple example of this sort of unknown sizing is as follows:

Listing 2.1: Example 1 - failure of strcpy()

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

int main( int argc, char **argv )
{
   char name[10];

   strcpy(name, "This␣is␣a␣long␣name");

   printf("Name␣=␣%s\n",name);

   return 0;
}
```

On compiling and running this program, possible output could be as follows:

```
bash-2.05\$ ./example1

Name = This is a long name
```

```
Segmentation fault

bash-2.05\$
```

Whilst the output from the program looks “normal” (apart from the Segfault of course), this code exhibits undefined behaviour due to using a pointer outside of a vector of allocation. It’s important to understand that *undefined* means exactly that, *undefined*. There’s no gaurantee of a segfault or other predictable behaviour, hence why debugging such bugs can be so annoying.

The effect here is that although *strcpy()* will result in the full valid string being written to memory, if the destination buffer is too small, *strcpy()* will commence writing beyond the buffer, possibly overwriting other critical memory locations.

It should be *strongly* emphasised that this program won’t always Segfault, it’s an intermittent failure based on many factors. The point though is that the program is overwriting memory it should not be and thus will potentially cause issues. Many times a program will compile and run without any ill-effect but intermittently it will produce invalid results or segfaulting. This intermittent behaviour makes debugging buffer overflows rather difficult.

If you must use *strcpy()*, then there are a few precautions you can take to at least minimise the potential of a buffer overflow, namely by strictly enforcing the length of the source string (Note - there’s nothing you can do about it if the input string is immutable)

Listing 2.2: Example 2 - Getting around strcpy()’s limitations

```c
/*
 * Safe usage of strcpy() - though still not
 * recommended
 *
 */

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

#define NAME_STR_LEN 10

int main( int argc, char **argv )
{
   char name[NAME_STR_LEN +1];
   char inputname[]="This␣is␣a␣long␣name";

   // If we really must use strcpy, then at least
   // force a test and termination of the input string

   if (strlen(inputname) > NAME_STR_LEN)
   {
      inputname[NAME_STR_LEN] = '\0';
   }

   strcpy(name, inputname);

   printf("Name␣=␣%s\n",name);
   return 0;
:}
```

2.2 strncpy

The public's favorite replacement for *strcpy()* has been announced as *strncpy()*, a function which we supposedly should have used from the start, the function which prevents buffer overflows, because it entails a size limit as one of its parameters... well, not quite.

The function won't overwrite beyond the bounds of what you set (via the size parameter), but *strncpy* does still hold a nasty little secret. As quoted directly from the Linux supplied man pages [man strncpy]:

> " Thus, if there is no null byte among the first n bytes of src, the result wil not be null-terminated."

Yes, even the missing 'l' is right.

Even though *strncpy()* won't directly cause a buffer overflow, the usage of the string it writes to the destination buffer can.

Listing 2.3: Example of strncpy() failure

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

#define FOO_STR_LEN 10

int main( int argc, char **argv )
{
   char foo[FOO_STR_LEN +1];

   // Copy our oversized string to foo, but this wont
   // cause a buffer overflow here

   strncpy(foo,"This␣is␣a␣long␣string",FOO_STR_LEN);

   // But printing out the resultant string will cause
   // undefined results because the string 'foo' does not
   // have a terminating \0 !!!

   printf("Foo␣=␣%s\n",foo);

   return 0;
}
```

Running this program will result in a similar output to what follows:

```
bash-2.05\$ ./example3

Foo = This is a ÿ¿wùÿ¿ëc@`

bash-2.05\$
```

Hence, we can see that although the strncpy function call did correctly limit the copying to the destination, but (through it's design) it failed to terminate the string with a \0. Once again, I exert, this is the way *strncpy()* is supposed to work. It is only due to the widespread usage of *strncpy()* as a supposedly "buffer safe" *strcpy()* replacement that *strncpy()* is also about to walk itself into the halls of buffer overflow fame.

Fortunately, there is an additional *str*cpy()* function which does do what people expect, *strlcpy()*, however, *strlcpy()* is not yet widely implemented.

As an interim replacement, the following source code is suggested:

Listing 2.4: strncpy replacement source code

```
char *CP_strncpy (char *dst, const char *src, size_t len)
{
      // If we have no buffer space, then it's futile attempting
      //    to copy anything, just return NULL
      if (len==0) return NULL;

      // Providing our destination pointer isn't NULL, we can
      //    commence copying data across

      if (dst)
      {
         char *dp = dst;

         // If our source string exists, start moving it to the
         //    destination string character at a time.
         if (src)
         {
            char *sp = (char *)src;
            while ((--len)&&(*sp)) { *dp=*sp; dp++; sp++; }
         }

         *dp='\0';
      }

   return dst;
}
```

2.3 strtok

Few functions come close to *strtok()* when it comes to producing the most obscure, yet, seemingly perfectly legal results (that is to say, it does not cause segfaults immediately). The problem with strtok is that it has a single shared working buffer (where it stores the string which it is processing). Hence, you cannot perform two independent *strtok()* processes concurrently or nested. The program listed here provides a good example of what happens when you nest two *strtok()* processes.

Listing 2.5: strtok failure in nested requests

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

```
#include <string.h>

int main( int argc, char **argv )
{
   char foo[]="This␣is␣my␣string␣which␣I␣want␣parsed";
   char bar[]="-I␣-also␣-want␣-this␣-string␣-parsed";
   char *foo_token, *bar_token;

   printf("Foo␣String␣=␣’%s’\nBar␣String␣=␣’%s’\n", foo, bar);
   foo_token = strtok(foo,"␣");

   printf("Foo␣token␣1␣=␣’%s’\n",foo_token);

   foo_token = strtok(NULL,"␣");

   printf("Foo␣token␣2␣=␣’%s’\n",foo_token);

   // Now lets start decoding our 2nd string, bar

   bar_token = strtok(bar,"␣");

   printf("Bar␣token␣1␣=␣’%s’\n",bar_token);

   foo_token = strtok(NULL,"␣");

   printf("Foo␣token␣3␣=␣’%s’\n",foo_token);

   return 0;

}
```

On running the compiled program, our output looks like:

```
Foo String = ’This␣is␣my␣string␣which␣I␣want␣parsed’
Bar String = ’-I␣-also␣-want␣-this␣-string␣-parsed’
Foo token 1 = ’This’
Foo token 2 = ’is’
Bar token 1 = ’-I’
Foo token 3 = ’-also’
```

As we can see, the second strtok call with a string initialisation (with the string *bar*) causes the work space for the foo strtok sequence to be over-written. In the example, it’s obvious to see the conflict, however, a more problematic situation arises if you’re strtok’ing and calling another function which might possibly call *strtok()* as well. This causes much confusion, not to mention many hours wasted wondering if the debugger is broken.

A replacement for *strtok()* is within the reach of most programmers, in fact, I suggest that every programmer does endeavor to write their own. Below is the listing of a pair of source files which implement a replacement *strtok()* function. Take note that there’s a third parameter with the replacement *strtok()* implementation, this parameter ensures that concurrent or nested strtok type calls will not interferre with their peer’s workspaces. The source code presented here is not guaranteed to be “optimal”, certainly there are many possibilities for optimising this function, however, our focus is to provide a working function *first*.

Listing 2.6: Safe replacement for strtok

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

/**
 **  Define the structure we need for strtok.
 ** We need to carry this information with us so that
 ** this strtok replacement will be thread safe, as
 **/
struct CP_strtok
{
   char *start;
   char delimeter;
};


char *CP_strtok( struct CP_strtok *st, char *line, char *
    delimeters )
{
   char *stop;
   char *dc;
   char *result = NULL; /** The pointer we will return **/

   /** If the line parameter is non-null, then we set the start of
    ** our token to be the start of the line
    **/
   if ( line )
   {
      st->start = line;
   }

   /** strtok behaviour clone:
    ** Strip off any leading delimeters
    **/
   dc = delimeters;
   while ((st->start)&&(*dc != '\0'))
   {
      if (*dc == *(st->start))
      {
         st->start++;
         dc = delimeters;
      }
      else dc++;
   }

   /** After the delimeters (if any) have been passed,
    ** we are then left at the position in the string
    ** where the token will start.
    **
    ** Set the return value to point to this location.
    **/
   result = st->start;

   if ((st->start)&&(st->start != '\0'))
   {
      stop = strpbrk( st->start, delimeters ); /** locate our next
           delimeter **/
      /** If we found a delimeter, then that is good.
       ** We must now break the string here
       ** and don't forget to store the character
       ** which we stopped on.  Very useful bit
       ** of information for programs which process expressions.
       **/

      if (stop)
      {

         /** Store our delimeter. **/

         st->delimeter = *stop;

         /** Terminate our token at the location of the first
             delimeter **/

         *stop = '\0';


         /** Because we're emulating strtok() behaviour here, we
             have to
          ** absorb all the concurrent delimeters, that is, unless
               we
          ** reach the end of the string, we cannot return a
              string with
          ** no chars. **/

         stop++;
         dc = delimeters;
         while (*dc != '\0')
         {
            if (*dc == *stop)
            {
               stop++;
               dc = delimeters;
            }
            else dc++;
         } // While

         if (*stop == '\0') st->start = NULL;
         else st->start = stop;

      } else {
         st->start = NULL;
         st->delimeter = '\0';
      }

   } else  {
      st->start = NULL;
      result = NULL;
   }


   return result;
}
```

2.4 strcat / strncat

Both *strcat()* and *strncat()* are rather poor functions for use in an environment where keeping a restraint on buffer overflows is needed (which is basically everywhere). It's abundantly apparent that *strcat()*, with no size control parameter will obviously be an ideal candidate for buffer overflow, however, *strncat()*, whilst it does provide the facility of dictating how many chars to copy, offers it in such a way that's not entirely useful to us, yet, is quite often used in a way which is totally wrong.

With *strncat()*, the fault often lies (again) within the perception that the size parameter depicts the maximum size of the *destination* buffer. In fact, it depicts the maximum number of characters we want to copy from the *source*.

A good example of *strncat()* abuse is depicted in the following source code. Note how the size parameter of *strncat()* is incorrectly specified as the destination buffer size.

Listing 2.7: Example of poor strncat usage

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

#define FOO_STR_LEN 20

int main( int argc, char **argv )
{
   char foo[FOO_STR_LEN+1]="";
   char startstr[]="Start␣of␣the␣string";
   char endstr[]="...␣end␣of␣string";

   strncat(foo,startstr, FOO_STR_LEN);
   printf("Foo␣=␣'%s'\n",foo);

   strncat(foo,endstr, FOO_STR_LEN);
   printf("Foo␣=␣'%s'\n",foo);

   return 0;

}
```

The output of this program executing is typically like as follows:

```
bash-2.05$ ./example_strncat
Foo = 'Start␣of␣the␣string'
Foo = 'Start␣of␣the␣string...␣end␣of␣string'
Segmentation fault
bash-2.05$
```

A suitable replacement for *strcat()* and *strncat()* which provides the desired behaviour is included below. Once again, I emphasise, that *strncat()* is not so much a bad function, but rather it's often misused, as is the case with a lot of string functions. However, usage of *zstrncat()*, in place of *strncat()*, additionally reduces the complexity of your coding by eliminating the need to keep track of the space remaining in your buffer. This function could additionally be extended to take another parameter to return the end of string marker (of the destination), this can facilitate in the saving of CPU cycles when dealing with particularly long destination buffers and a lot of string catenation (Normally one would have to determine the end of string location by traversing the string from the start and looking for the \0 marker).

Listing 2.8: zstrncat - safe strcat and strncat replacement

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


char *zstrncat( char *dst, char *src, size_t len )
{
   char *dp = dst;
   char *sp = src;
   int cc;

   // Compensate for our need for a \0 at the end of the string

   len--;


   // Locate the end of the current string.
   cc = 0;
   while ((*dp)&&(cc < len)) { dp++; cc++; }

   // While we have more source, and there’s more char space left
       in the buffer

   while ((*sp)&&(cc < len))
   {
      *dp = *sp;
      dp++;
      sp++;
      cc++;
   }

   // Terminate dst, as a guarantee of string ending.
   // Only if the input length was 0 should we not do this
   // as a length of zero means there was no buffer space
       initially

   if (len >= 0)
   {
      *dp = ’\0’;
   }

   return dst;
}
```

2.5 sprintf

As with *strcpy()* and *strcat()*, *sprintf()* fails to place any buffer limits checking into its routines. Fortunately, the 'n' replacement, *snprintf()* is quite safe to use and exhibits no unusual behaviour. In fact, *snprintf()* could be used as a replacement for *strcpy()*/*strncpy()*.

Chapter 3

Memory related functions

Memory allocation functions, while fairly simple, do still come up as problematic, especially when dealing with reallocation. Other situations are rare, but should still be noted (for *malloc*). Additionally, there are issues regarding the interpretation of the ANSI specification, regarding what to do with *null* pointers.

3.1 realloc

Whilst *realloc()* does not exhibit unusual or complex behaviour (although what happens behind the scenes is quite complex), people can often fail to account for what happens when things do go wrong with the allocation request, ie, when realloc returns a *null* pointer. The following code demonstrates a typical sort of situation, where *realloc()* is used to provide an ever larger block of memory to cater for the catenation of a string to the end of an existing string.

Although it is quite obvious that this program will (eventually) consume all the available memory, what is not apparent is that you will be unable to free it once memory has run out.

Listing 3.1: realloc - example of typical, bad usage

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

int main( int argc, char **argv )
{
   char arbstring[]="This␣is␣an␣arbitrary␣string\n";
   char *foo;
   int blocksize=0;
   int slen;

   blocksize = slen = strlen(arbstring) +1;

   while (1)
   {
      foo = realloc(foo, blocksize);
      blocksize += slen;
      strcat(foo,arbstring);
   }

   return 0;
}
```

The reason why you will not be able to free the allocated block of memory (which is probably quite large) is due to the result from *realloc()* being immediately applied to the variable which we are using to keep track of the start of the memory block. When *realloc()* fails to allocate a memory block, it will return a *null* pointer, thus foo immediately points to *null*, and we no longer know where the (previous) block of memory which foo pointed to is, it's now a non recoverable block.

An example of how such a process should be coded is provided:

Listing 3.2: realloc - example of good, safe usage

```
int main( int argc, char **argv )
{
   char arbstring[]="This␣is␣an␣arbitrary␣string\n";
   char *foo, *bar;
   int blocksize=0;
   int slen;

   blocksize = slen = strlen(arbstring) +1;

   while (1)
   {
      bar = realloc(foo, blocksize);
      if (bar)
      {
         foo = bar;
         blocksize += slen;
         zstrncat(foo, arbstring, blocksize);

      }
      else {
         ... perform graceful exit
      }
   }

   return 0;
}
```

3.2 free

There is the occasional debate surrounding the free function. The problem lies with the situation of "Should I test a pointer for *null* before free'ing it?"

Answers can range from "No, ANSI says it's safe to *free(NULL)*.", to "Yes, despite ANSI, it's good practice to test the pointer before trying to free it", and of course, there's the inbetween.

The issues arise from a few points.

- Free'ing a *null* pointer causes a segmentation fault on some non-ANSI or just plain broken implementations
- Attempting to free a *null* pointer can be an indication that there are issues elsewhere in the code
- Redundant code makes for slower and larger programs

Certainly it is agreeable that if you are attempting to free a null pointer, then you're most certainly in a situation where somewhere prior in your code there is something amiss. However, that is not specifically a reason to not program in a defensive manner. As humans we most certainly do create mistakes, by testing our pointer before hand gives us another chance to detect a mistake before throwing the pin rather than the grenade.

As a simple example, see the following code:

Listing 3.3: Example of segmentation faulting using free

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

int main(int argc, char **argv)
{
   char *foo = NULL;

   free(foo);

   return 0;
}
```

On running this program, a segmentation fault may occur (on some implementations such as early linux glibc's).

Listing 3.4: Example of a graceful exit from freeing a null pointer

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

int main(int argc, char **argv)
{
   char *foo = NULL;

   if (!foo)
   {
      fprintf(stderr,"Error:%d:␣Attempting␣to␣free␣variable␣'foo'␣
          which␣is␣null.\n",__LINE__);
      exit(1);
   }
   else free(foo);

   return 0;
}
```

At least now when we run our program, the output resembles something a little more useful for debugging.

```
bash-2.05$ ./example_freegraceful
Error:10: Attempting to free variable 'foo' which is null.
bash-2.05$
```

3.3 malloc

The discussion for malloc is a simple one, don't assume that malloc will always return a non-null pointer. Although it's rare, it can still happen. Always test your returned value from malloc for null value before proceeding to use it.

3.4 sizeof

The function *sizeof()* is often misused, with good intentions, to return the size of a string or some other arbitrarily allocated block of memory. Due to *sizeof()* being a compile-time resolved function, it cannot return useful results for dynamically changing quantities.

As a rule, attempt to only use *sizeof()* on types, and not variables. For strings, use *strlen()*. Although there are many exceptions to this, it will at the very least prevent you from (generally) attempting to *sizeof()* a pointer.

Chapter 4

Input/Output related functions

4.1 gets

Many of the bad functions in C can be worked around by performing pre or post call checks on the variables. Unfortunately, *gets()* is impossible to protect yourself against. Consider the situation where you allocate a 1K buffer to read input from stdin, surely that's big enough for most inputs, that is until someone sends you 1K and 1 byte, your program has just had a buffer overflow. The fundamental problem with *gets()* is that there's no way to protect yourself from buffer overflows because there's no way of restricting how much input to accept while *gets()* is called. The word never is often abused but *never* use *gets()*. Use *fgets(buffer, buffer_size, stdin)* instead, please.

Listing 4.1: Example of why gets is dangerous

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

int main(int argc, char **argv)
{
   char buffer[20]; /** setup a buffer of 20 chars **/

   do {
      printf("Say␣something␣:␣");

      /** If the input is greater than 19 chars, this
        ** gets call will overflow.
        **
        ** Your program will not always segfault
        ** immediately but be assured, your program
        ** has now mangled something **/

      gets(buffer);
      printf("You␣said␣'%s'\n",buffer);
   } while ( strlen(buffer) > 0 );

   return 0;
}
```

```
$ ./gets-example
Say something : Hello
You said 'Hello'
Say something : This is a really long string which will mess
    things up
You said 'This␣is␣a␣really␣long␣string␣which␣will␣mess␣things␣up'
Say something :
You said ''
Segmentation fault
```

Notice that in this example, gets-example didn't segfault until the program went to exit. It should be noted that the behaviour of the program after a gets buffer overflow (or any overflow for that matter) is undefined, some will segfault/crash almost immediately, others will show no ill signs.

When you do replace all your gets() calls with fgets(), you'll still have one further thing to adjust. The gets() does not store the trailing \n character but fgets() does. A simple way of doing this is;

```
len = strlen(buffer);
if ((len >0)&&(buffer[len-1] == '\n'))
   buffer[len-1] = '\0';
```

The reason we have to check for the presence of the trailing \n is because *fgets()* may not always have read the entire input up to the terminating \n character. See section 4.2 for more details on possible *fgets()* issues.

4.2 fgets

It should be noted at first that *fgets()* is actually quite a good function. The reason for putting *fgets()* into this book is to ensure that the one potential mistake that can arise from using it is covered.

Be aware that there's no gaurantee that a call to *fgets()* will always return a complete line to the terminating
n character. If your buffer is just one char too small, the next call to *fgets()* will return a string containing only the single
n character. This type of behaviour may be misleading in situations where your program is seeking for a delimiting 'blank line', as such is the case when parsing MIME or HTTP data.

4.3 scanf

For user interaction and input, *scanf()* has often been used, however, it's problems lie within the fragile nature of the format of data, as specified by the format string. Many problems can occur from failing to correctly input all data (causing a spill over into the next *scanf()* call).

Another issue with *scanf()* is the lack of buffer limits when performing a string read/translation makes it as fatal as *gets()*.

Listing 4.2: Example of why scanf can be dangerous

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

int main( int argc, char **argv )
{
   char buf[10];

   printf("Enter␣your␣name:␣");
   scanf("%s",buf);
   printf("\nYour␣name␣is␣%s", buf);

   return 0;

}
```

```
worker@workbox:/extra/shared/documents/c-of-peril/examples
$ ./scanf-example
Enter your name: Paul

Your name is Paul


$ ./scanf-example
Enter your name: sdflkhjlh3tlh23l;
    th20f3h909ch029hd092h30hd09h309h309hd3hdf

Segmentation fault

worker@workbox:/extra/shared/documents/c-of-peril/examples
```

Buffer overflows on the output string parameters can be avoided by explicitly defining size limits in the *scanf()* format. This is shown in the following code. Note the buffer size prefix in the %s format specifier.

Listing 4.3: Example of how to buffer size limit in scanf

```
   char buf[10];

   printf("Enter␣your␣name:␣");
   scanf("%9s",buf);
   printf("\nYour␣name␣is␣%s", buf);
```

Rather, use *fgets(buffer, buffer_size, input_file)*, this ensures that everything, including the line termination is read into the buffer *buffer* (assuming *buffer* is large enough to hold the potential input)

4.4 sprintf, fprintf, syslog and other output related functions

As C imposes very loose restrictions on what you can make it do, there are possible situations which can cause data corruption or even segmentation faulting through the non conformance of types specifed in the output format, versus those actually supplied. Sometimes, it can even mean your debugging reporting output can be the cause of a segfault.

The following source code, when compiled and run will cause a segfault:

Listing 4.4: Example of generating a segfault through the use of wrongly matched format, value pairs

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

int main(int argc, char **argv)
{
   int foo=3254325;

   syslog(1,"This␣is␣an␣integer␣%s",foo);

   return 0;
}
```

In this instance, the segmentation fault is a result of the program interpreting the integer value *foo* as an address of a character pointer (the value of *foo* was interpreted as a pointer value). Incidently, had the typing of the variables been reversed, such that *foo* was a string and the syslog was expecting an integer, then so long as the string was sufficiently long (*sizeof(int)* bytes, inclusive of the terminating \0 byte) we would get results that did not match what were expected.

Mostly this example serves to remind that one should not take lightly the warnings which the C compiler will produce when attempting to compile such code. ie, the GNU gcc compiler compiled with the following error:

```
example_syslog.c: In function 'main':

example_syslog.c:9: warning: format argument is not a pointer (arg
     3)
```

In a large compile, such warnings can be overlooked and be potentially fatal. Just because a C program compiles does not imply that it is going to function in a valid manner.

Chapter 5

Macro Mishaps

5.1 Semicolon at the end of macros

Semicolons should not generally be placed at the end of macros. The preprocessor replaces the text as-is, thus an additional semicolon usually leads to syntactical errors. For example:

```
#define SIZE_LIMIT 128;
```

```
if (strlen(foo) > SIZE_LIMIT)
```

which converts to:

```
if (strlen(foo)> 128;)
```

Compilation of this code will most likely result in an error:

```
parse error before `;'
```

5.2 Macro name and left bracket separation

There cannot be any white space (spaces or tabs) between the macro name and the first left bracket (for parameters). This is due to the preprocessor interpreting the occurrence of white space as the end of a macro name (with no parameters). For example:

```
#define min(x,y) (x<y)?x:y
```

causing all occurrences of 'min' to be replaced with:

```
(x,y) (x<y)?x:y
```

Most occurances of this type of error will induce a compilation error/warning.

5.3 Operator precedence for parameters

Find something for here...

5.4 Operator precedence for expressions

Because macros are text substitutions prior to a compiling process, no order of precedence checking is performed, arguments are just substituted, consequently the resulting code substitution is not guaranteed to have the same order of precedence as intended (by the programmer). The following example demonstrates this.

Listing 5.1: Example exhibiting incorrect order of precedence due to macro expansion effect

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

#define sum(x,y) x + y

int main( void )
{
   int a = 10;
   int b = 3;
   int c;

   c = 2 *sum(a,b);

   printf("2␣*␣(%d␣+␣%d)␣=␣%d\n", a, b, c );

   return 0;
}
```

5.5 Multiple evaluation of parameters

A good example is if we were attempting to print out a list of the cube of the integers from 1 to 10.

Listing 5.2: Example of code incorrectly using a macro causing undesired side effects

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

#define cube(x) (x)*(x)*(x)

int main( void )
{
   int i = 1;

   while (i <= 10)
   {
      printf("Cube␣of␣%d␣=␣%d\n", i, cube(i++) );
   }

   return 0;
}
```

On executing this small program, our output is completely different from what we initially would anticipate.

Listing 5.3: Output of macro example

```
Cube of 4 = 1
Cube of 7 = 64
Cube of 10 = 343
Cube of 13 = 1000
```

There is no real 'cure' for this condition other than to avoid performing operations within the parameters of the macro. In the case of the presented example, moving the *i++* operation outside of the macro will result in correct operation.

5.6 Single statements versus multiple statements.

In macros that expand to multiple statements and later used in conditional/loop constructs:

```
#define foo(x) bar(x); baz(x);
```

```
...
```

```
if (test(x)) foo(x);
```

which will expand to:

```
if (test(x)) bar(x); baz(x);
```

- the *baz(x)* call should be inside the conditional block but is not.

The solution:

```
#define foo(x) { bar(x); baz(x); }
```

solves the problem:

```
if (test(x)) foo(x);
```

which gives:

```
if (test(x)){ bar(x); baz(x); };
```

Chapter 6

Optimising

Most likely this part of the book will not provide you with what you want to read, or care to hear. Let's be utterly blunt, a majority of so called optimisations of code are completely pointless, other than to gain you entry into the IOCCC (Obfusicated Code Competition). Further to the point, programming, *especially* C programming is not the place to express creativity or to have a lack of constraints on egotistical flair.

This part of the book is going to provide examples of optimising going wrong either in maintaince phase or where speed/space gains are actually attributed to loss of corner cases or utterly incorrectly logic.

Again, avoid code level optimisations, look rather to algorithm and design optimisations.

To be appended to...

6.1 A Matter of Style

Unlike languages such as FORTRAN and Python which impose rules about formatting and white space, C can be formatted in a profoundly stupid number of ways. This flexibility leads to quite a variation in styles, all hotly contested to be 'the best'. Some programmers however take their desires to optimise a little too far and start attacking the actual source code. There is little reason to cull characters from a source file for the sake of size when the losses to readability and maintainability are magnitudes worse.

To be appended to... More examples from #c perhaps...

Chapter 7

Syntax

When writing C programs, it's quite common to write something which means something different to the compiler than what you are expecting. To make matters worse, the compiler will, unless it perceives what you created as being an error, will happily compile it.
To be appended to...

Chapter 8

Links

Naturally not everything can be covered in one book. This book especially does not attempt to be a singular reference for all C. Listed below are a series of links which may prove useful to readers for further information or resources.

- Replacement Libraries
  - Better String Library, http://bstring.sourceforge.net/
- Debugging and runtime analysis tools
  - *Valgrind, http://valgrind.kde.org/*
    Enough praise for Valgrind simply cannot be said. If there's one tool to make you own a linux box, this is it. Valgrind will help you find those ever so annoying memory leaks, fixing many intermittent problems without you explicitly realising you've fixed them (okay, so that can sometimes be a bad thing). What makes this tool so useful is that it's incredibly easy to use. You simply prefix your normal command line invocation of your program with (eg) *valgrind -v –tool=memcheck* and behold, you'll most likely be greeted with many points of code leaking memory.
  - *ltrace, http://packages.debian.org/unstable/utils/ltrace.html*
    Tracks runtime library calls in dynamically linked programs ltrace is a debugging program which runs a specified command until it exits. While the command is executing, ltrace intercepts and records the dynamic library calls which are called by the executed process and the signals received by that process. It can also intercept and print the system calls executed by the program. The program to be traced need not be recompiled for this, so you can use it on binaries for which you don't have the source handy. You should install ltrace if you need a sysadmin tool for tracking the execution of processes.
  - *strace, http://www.liacs.nl/ wichert/strace/*
    Strace is a system call trace, i.e. a debugging tool which prints out a

trace of all the system calls made by a another process/program. The program to be traced need not be recompiled for this, so you can use it on binaries for which you don't have source.

System calls and signals are events that happen at the user/kernel interface. A close examination of this boundary is very useful for bug isolation, sanity checking and attempting to capture race conditions

- *dtrace, http://www.sun.com/bigadmin/content/dtrace/*
  DTrace is a comprehensive dynamic tracing framework for the Solaris Operating Environment. DTrace provides a powerful infrastructure to permit administrators, developers, and service personnel to concisely answer arbitrary questions about the behavior of the operating system and user programs.

Appendix A

Book Development Notes

A.1 CHANGELOG / ERRATA

- *Monday, November 29, 2004 - 11H56, Paul L Daniels*
  Added valgrind (linux), ltrace (linux), strace, dtrace (solaris10) to Links section

A.2 AUTHORS

- *Paul L Daniels, pldaniels@pldaniels.com*
  Principle author of *C of Peril*.