Common C Programming Errors

Table of Contents

1. Introduction

This document lists the common C programming errors that the author sees time and time again. Solutions to the errors are also presented.

Another great resource is the C FAQ. Gimpel Software also has a list of hard to detect C/C++ bugs that might be useful.

2. Beginner Errors

These are errors that beginning C students often make. However, the professionals still sometimes make them too!

2.1 Forgetting to put a `break` in a `switch` statement

Remember that C does not break out of a `switch` statement if a `case` is encountered. For example:

```
int x = 2;
switch(x) {
case 2:
  printf("Two\n");
case 3:
  printf("Three\n");
}
```

prints out:

```
Two
Three
```

Put a break to break out of the switch:

```
int x = 2;
switch(x) {
case 2:
  printf("Two\n");
  break;
case 3:
  printf("Three\n");
  break;   /* not necessary, but good if additional cases are added later */
}
```

2.2 Using = instead of ==

C's = operator is used exclusively for assignment and returns the value assigned. The == operator is used exclusively for comparison and returns an integer value (0 for *false*, not 0 for *true*). Because of these return values, the C compiler often does not flag an error when = is used when one really wanted an ==. For example:

```
int x = 5;
if ( x = 6 )
  printf("x equals 6\n");
```

This code prints out `x equals 6`! Why? The assignment inside the `if` sets `x` to 6 and returns the value 6 to the `if`. Since 6 is not 0, this is interpreted as *true*.

One way to have the compiler find this type of error is to put any constants (or any r-value expressions) on the left side. Then if an = is used, it will be an error:
`if ( 6 = x)`

2.3 scanf() errors

There are two types of common `scanf()` errors:

2.3.1 Forgetting to put an ampersand (&) on arguments

`scanf()` must have the address of the variable to store input into. This means that often the ampersand address operator is required to compute the addresses. Here's an example:

```
int x;
char * st = malloc(31);

scanf("%d", &x);    /* & required to pass address to scanf()    */
scanf("%30s", st); /* NO & here, st itself points to variable! */
```

As the last line above shows, sometimes no ampersand is correct!

2.3.2 Using the wrong format for operand

C compilers do *not* check that the correct format is used for arguments of a `scanf()` call. The most common errors are using the `%f` format for doubles (which must use the `%lf` format) and mixing up `%c` and `%s` for characters and strings.

2.4 Size of arrays

Arrays in C always start at index 0. This means that an array of 10 integers defined as:

```
int a[10];
```

has valid indices from 0 to 9 *not* 10! It is very common for students go one too far in an array. This can lead to unpredictable behavior of the program.

2.5 Integer division

Unlike Pascal, C uses the / operator for both real and integer division. It is important to understand how C determines which it will do. If both operands are of an integal type, integer division is used, else real division is used. For example:

```
double half = 1/2;
```

This code sets `half` to 0 not 0.5! Why? Because 1 and 2 are integer constants. To fix this, change at least one of them to a real constant.

```
double half = 1.0/2;
```

If both operands are integer variables and real division is desired, cast one of the variables to `double` (or `float`).

```
int x = 5, y = 2;
double d = ((double) x)/y;
```

2.6 Loop errors

In C, a loop repeats the very next statement after the loop statement. The code:

```
int x = 5;
while( x > 0 );
  x--;
```

is an infinite loop. Why? The semicolon after the `while` defines the statement to repeat as the null statement (which does nothing). Remove the semicolon and the loop works as expected.

Another common loop error is to iterate one too many times or one too few. Check loop conditions carefully!

2.7 Not using prototypes

Prototypes tell the compiler important features of a function: the return type and the parameters of the function. If no prototype is given, the compiler *assumes* that the function returns an int and can take any number of parameters of any type.

One important reason to use prototypes is to let the compiler check for errors in the argument lists of function calls. However, a prototype *must* be used if the function does not return an int. For example, the `sqrt()` function returns a double, not an int. The following code:

```
double x = sqrt(2);
```

will not work correctly if a prototype:

```
double sqrt(double);
```

does not appear above it. Why? Without a prototype, the C compiler assumes that `sqrt()` returns an int. Since the returned value is stored in a double variable, the compiler inserts code to convert the value to a double. This conversion is not needed and will result in the wrong value.

The solution to this problem is to include the correct C header file that contains the `sqrt()` prototype, `math.h`. For functions you write, you must either place the prototype at the top of the source file or create a header file and include it.

2.8 Not initializing pointers

Anytime you use a pointer, you should be able to answer the question: *What variable does this point to?* If you can not answer this question, it is likely it doesn't point to *any* variable. This type of error will often result in a `Segmentation fault/coredump` error on UNIX/Linux or a general protection fault under Windows. (Under good old DOS (ugh!), anything could happen!)

Here's an example of this type of error.

```
#include <string.h>
int main()
{
  char * st;   /* defines a pointer to a char or char array */
```

```
    strcpy(st, "abc");  /* what char array does st point to?? */
    return 0;
  }
```

How to do this correctly? Either use an array or dynamically allocate an array.

```
  #include <string.h>
  int main()
  {
    char st[20];   /* defines an char array */

    strcpy(st, "abc");  /* st points to char array */
    return 0;
  }
```

or

```
  #include <string.h>
  #include <stdlib.h>
  int main()
  {
    char *st = malloc(20);   /* st points to allocated array*/

    strcpy(st, "abc");  /* st points to char array */
    free(st);               /* don't forget to deallocate when done! */
    return 0;
  }
```

Actually, the first solution is much preferred for what this code does. Why? Dynamical allocation should only be used when it is required. It is slower and more error prone than just defining a normal array.

3. String Errors

3.1 Confusing character and string constants

C considers character and string constants as very different things. Character constants are enclosed in *single quotes* and string constants are enclosed in *double quotes*. String constants act as a pointer to the actually string. Consider the following code:

```
  char ch = 'A';      /* correct */
  char ch = "A";      /* error   */
```

The second line assigns the character variable `ch` to the address of a string constant. This should generate a compiler error. The same should happen if a string pointer is assigned to a character constant:

```
  const char * st = "A";      /* correct */
  const char * st = 'A';      /* error   */
```

3.2 Comparing strings with ==

Never use the == operator to compare the value of strings! Strings are `char` arrays. The name of a `char` array acts like a pointer to the string (just like other types of arrays in C). So what? Consider the following code:

```
  char st1[] = "abc";
  char st2[] = "abc";
  if ( st1 == st2 )
    printf("Yes");
  else
    printf("No");
```

This code prints out *No*. Why? Because the == operator is comparing the *pointer values* of `st1` and `st2`, not the data pointed to by them. The correct way to compare string values is to use the `strcmp()` library function. (Be sure to include `string.h`) If the `if` statement above is replaced with the following:

```
if ( strcmp(st1,st2) == 0 )
  printf("Yes");
else
  printf("No");
```

the code will print out *Yes*. For similar reasons, don't use the other relational operators (<,>, *etc.*) with strings either. Use `strcmp()` here too.

3.3 Not null terminating strings

C assumes that a string is a character array with a terminating null character. This null character has ASCII value 0 and can be represented as just `0` or `'\0'`. This value is used to mark the end of meaningful data in the string. If this value is missing, many C string functions will keep processing data past the end of the meaningful data and often past the end of the character array itself until it happens to find a zero byte in memory!

Most C library string functions that create strings will always properly null terminate them. Some do not (*e.g.*, `strncpy()`). Be sure to read their descriptions carefully.

3.4 Not leaving room for the null terminator

A C string must have a null terminator at the end of the meaningful data in the string. A common mistake is to not allocate room for this extra character. For example, the string defined below

```
char str[30];
```

only has room for only 29 (not 30) actually data characters, since a null *must* appear after the last data character.

This can also be a problem with dynamic allocation. Below is the correct way to allocate a string to the exact size needed to hold a copy of another.

```
char * copy_str = malloc( strlen(orig_str) + 1);
strcpy(copy_str, orig_str);
```

The common mistake is to forget to add one to the return value of `strlen()`. The `strlen()` function returns a count of the data characters which does *not* include the null terminator.

This type of error can be very hard to detect. It might not cause any problems or only problems in extreme cases. In the case of dynamic allocation, it might corrupt the *heap* (the area of the program's memory used for dynamic allocation) and cause the *next* heap operation (`malloc()`, `free()`, *etc.*) to fail.

4. Input/Output Errors

4.1 Using `fgetc()`, *etc.* incorrectly

The `fgetc()`, `getc()` and `getchar()` functions all return back an *integer* value. For example, the prototype of `fgetc()` is:

```
int fgetc( FILE * );
```

Sometimes this integer value is really a simple character, but there is one very important case where the return value is **not** a character!

What is this value? `EOF` A common misconception of students is that files have a special EOF character at the end. There is no special character stored at the end of a file. EOF is an *integer* error code returned by a function. Here is the **wrong** way to use `fgetc()`:

```
int count_line_size( FILE * fp )
{
  char ch;
  int  cnt = 0;

  while( (ch = fgetc(fp)) != EOF && ch != '\n')
    cnt++;
  return cnt;
}
```

What is wrong with this? The problem occurs in the condition of the `while` loop. To illustrate, here is the loop rewritten to show what C will do behind the scenes.

```
while( (int) ( ch = (char) fgetc(fp) ) != EOF && ch != '\n')
  cnt++;
```

The return value of `fgetc(fp)` is cast to `char` to store the result into `ch`. Then the value of `ch` must be cast back to an `int` to compare it with `EOF`. So what? Casting an `int` value to a `char` and then back to an `int` may not give back the original `int` value. This means in the example above that if `fgetc()` returns back the `EOF` value, the casting may change the value so that the comparison later with `EOF` would be false.

What is the solution? Make the `ch` variable an `int` as below:

```
int count_line_size( FILE * fp )
{
  int ch;
  int  cnt = 0;

  while( (ch = fgetc(fp)) != EOF && ch != '\n')
    cnt++;
  return cnt;
}
```

Now the only hidden cast is in the second comparison.

```
while( (ch = fgetc(fp)) != EOF &&  ch != ((int) '\n') )
  cnt++;
```

This cast has no harmful effects at all! So, the moral of all this is: **always** use an `int` variable to store the result of the `fgetc()`, `getc()` and `getchar()`.

4.2 Using `feof()` incorrectly

There is a wide spread misunderstanding of how C's `feof()` function works. Many programmers use it like Pascal's `eof()` function. However, C's function works differently!

What's the difference? Pascal's function returns true if the *next* read will fail because of end of file. C's function returns true if the *last* function failed. Here's an example of a misuse of `feof()`:

```
#include <stdio.h>
int main()
{
  FILE * fp = fopen("test.txt", "r");
  char line[100];

  while( ! feof(fp) ) {
    fgets(line, sizeof(line), fp);
    fputs(line, stdout);
  }
  fclose(fp);
  return 0;
}
```

This program will print out the last line of the input file *twice*. Why? After the last line is read in and printed out, `feof()` will still return 0 (false) and the loop will continue. The next `fgets()` fails and so the `line` variable holding the contents of the last line is not changed and is printed out again. After this, `feof()` will return true (since `fgets()` failed) and the loop ends.

How should this fixed? One way is the following:

```
#include <stdio.h>
int main()
{
  FILE * fp = fopen("test.txt", "r");
  char line[100];

  while( 1 ) {
    fgets(line, sizeof(line), fp);
    if ( feof(fp) )    /* check for EOF right after fgets() */
      break;
    fputs(line, stdout);
  }
  fclose(fp);
  return 0;
}
```

However, this is not the best way. There is really no reason to use feof() at all. C input functions return values that can be used to check for EOF. For example, fgets returns the NULL pointer on EOF. Here's a better version of the program:

```
#include <stdio.h>
int main()
{
  FILE * fp = fopen("test.txt", "r");
  char line[100];

  while( fgets(line, sizeof(line), fp) != NULL )
    fputs(line, stdout);
  fclose(fp);
  return 0;
}
```

The author has yet to see any student use the feof() function correctly!

Incidently, this discussion also applies to C++ and Java. The eof() method of an istream works just like C's feof().

4.3 Leaving characters in the input buffer

C input (and output) functions buffer data. Buffering stores data in memory and only reads (or writes) the data from (or to) I/O devices when needed. Reading and writing data in big chunks is much more efficient than a byte (or character) at a time. Often the buffering has no effect on programming.

One place where buffering is visible is input using scanf(). The keyboard is usually line buffered. This means that each line input is stored in a buffer. Problems can arise when a program does not process all the data in a line, before it wants to process the next line of input. For example, consider the following code:

```
int x;
char st[31];

printf("Enter an integer: ");
scanf("%d", &x);
printf("Enter a line of text: ");
fgets(st, 31, stdin);
```

The fgets() will not read the line of text that is typed in. Instead, it will probably just read an empty line. In fact, the program will not even wait for an input for the fgets() call. Why? The scanf() call reads the characters needed that represent the integer number read in, but it leaves the '\n' in the input buffer. The fgets() then starts reading data from the input buffer. It finds a '\n' and stops without needing any additional keyboard input.

What's the solution? One simple method is to read and dump all the characters from the input buffer until a '\n' after the scanf() call. Since this is something that might be used in lots of places, it makes sense to make this a function. Here is a function that does just this:

```
/* function dump_line
*  This function reads and dumps any remaining characters on the current input
*  line of a file.
*  Parameter:
*     fp - pointer to a FILE to read characters from
*  Precondition:
*     fp points to a open file
*  Postcondition:
*     the file referenced by fp is positioned at the end of the next line
*     or the end of the file.
*/
void dump_line( FILE * fp )
{
  int ch;

  while( (ch = fgetc(fp)) != EOF && ch != '\n' )
    /* null body */;
}
```

Here is the code above fixed by using the above function:

```
int x;
char st[31];

printf("Enter an integer: ");
scanf("%d", &x);
dump_line(stdin);
printf("Enter a line of text: ");
fgets(st, 31, stdin);
```

One incorrect solution is to use the following:

```
fflush(stdin);
```

This will compile but its behavior is undefined by the ANSI C standard. The `fflush()` function is only meant to be used on streams open for output, not input. This method does seem to work with some C compilers, but is completely unportable! Thus, it should not be used.

4.4 Using the `gets()` function

Do not use this function! It does not know how many characters can be safely stored in the string passed to it. Thus, if too many are read, memory will be corrupted. Many security bugs that have been exploited on the Internet use this fact! Use the `fgets()` function instead (and read from `stdin`). But remember that unlike `gets()`, `fgets()` does *not* discard a terminating `\n` from the input.

The `scanf()` functions can also be used dangerously. The `%s` format can overwrite the destination string. However, it can be used safely by specifying a width. For example, the format `%20s` will not read more than 20 characters.

5. Acknowlegements

The author would like to thank Stefan Ledent for suggesting the section on "Not leaving room for the null terminator"

Maintainer: Paul Carter (email: pacman128@gmail.com)

Last Updated: Tue Feb 19 18:00:33 EST 2008