C Keys

Ed Cashin, December 1999

---

abstract

The C programming language can be learned from many available resources, but often a few key points are neglected. These important yet often neglected points are the subject of this paper.

The tone of this paper is arrogant and pedantic in order to increase readability. The points covered in this paper are difficult to articulate, and that is perhaps one reason why many books and other resources that teach C neglect to cover them. Despite the tone, the author welcomes comments and corrections.

In a pinch the "precise and terse yet ugly" approach to conveying meaning has been used. The effect of this strategy is that the reader may have to read any given passage or sentence more than once.

Contents

1 no buffer overflows

C is famous for several things including its speed, its portability, its widespread use, and buffer overflows. Countless security problems have been caused by buffer overflow bugs.

It is not OK to release a program that does not handle unexpected situations gracefully. In exceptional circumstances a well-written program produces helpful warnings or errors, exiting with an error code if necessary. Letting a confused program run out of control until the operating system has to kill it is bad style, is dangerous, and leaves the user in a lurch.

What it comes down to in practice is this: being careful to write only to memory that has been properly allocated (and not yet freed).[1] And what that means is:

> *Do not copy data from a source of undetermined size into a finite region of allocated memory. (like the* `gets` *and* `scanf` *functions do)*[2]
>
> *Do not copy more data than can fit into the destination region of memory.*

1.1 gets

"`gets` is evil," is a phrase often heard on usenet. Using the `gets` function to read an unknown quantity of data is a bug. Reading input from standard input into a location in memory with `gets` is an example of such a bug. The reason is that `gets` provides no way at all to control the number of characters that are transfered.

An easy way to tell the quality of a book teaching something in C is to look for `gets`. If the author uses `gets`, that probably means that either:

1. The book is useless because the author knows the subject but is knowingly using poor code for purposes of illustration, or . . .
2. The book is useless because the author is a loser.

[1] Memory is allocated *automatically*, *statically*, and *dynamically*, in the case of function-local variables, the case of function-static and global variables, and the case of explicit requests for memory from the heap, respectively.

[2] If, instead of using `gets`, the programmer uses `fgets` to read in a line, control over the amount of copied data is maintained during input. At that point, `sscanf` (notice the extra 's') can do formatted reading safely.

1.2 related dangers

Other things to watch out for:

1. When using arrays, don't write to `array[arraysize]` (that's the element *after* the last element in the array), or to any other areas outside of the allocated array.
2. When using pointers, do not write to ...
   a. `NULL` pointers
   b. uninitialized pointers (pointers that aren't yet pointing to any allocated memory)
   c. pointers that point to freed memory
   d. or pointers that point to bits of memory outside of the allocated area

Doing those things often results in segfaults (if the operating system notices that you're writing to places you shouldn't write to) or in strange problems with malloc and free (if the O.S. can't tell you're doing something wrong).

2 declaration versus definition

The distinction between declaration and definition is only made by K&R and a few others. It is hugely important and largely ignored.

Declaration Declaration is process by which the programmer tells the compiler (and any humans watching) what's going on.

Definition Definition is the process by which the programmer ensures memory allocation; i.e. the programmer tells the compiler that something needs to be in the computer's memory.

Here are a couple of examples:

```
extern int errno;    /* declares that "errno" is an integer
                        that is defined elsewhere */

int i;               /* declares that "i" is an integer and
                        simultaneously defines it (compiler
                        provides memory space for one
                        integer) */
```

... a different but similar usage of the terms:

```
void greet(const char* s);      /* declares the function
                                   "greet", describing to the compiler
                                   what the function is like */

void greet(const char* s)           /* defines function */
{                                   /* "greet" */
```

```
    printf("hello, %s.\n", s);
}
```

These terms as distinguished here are not used consistently in C literature, but the distinction itself is the important thing.

3 identifiers versus values

Identifier A string of characters in the text of C source that the compiler associates (and your programming buddies associate) uniquely with an object in memory.

Value The contents of an object in memory determine the value of that object.

Variable The object in memory that can hold values.

Here, the word "object" has a meaning that predates Object Oriented Programming. Larry Wall calls this kind of object a "thingy." It is just a chunk of memory. We presume that this particular chunk is interesting in some way.

For example, I'm thinking of a particular chunk in most C programs that is often a four-byte sequence in memory. The contents of that memory chunk represent the integer specifying the number of command line arguments. This "representation in memory" provides a meaningful value for us (programmers and the compiler) to use. In the example I'm thinking of, the memory object is marked by the identifier, `argc`.

This distinction: *The string of characters, 'argc,' is different but related to the value (e.g. 3) in the memory object that is identified by 'argc';* becomes critical when pointers enter the picture. With this distinction clearly defined, pointers are easy to use; without it, pointers are downright scary.

Take the following code as an example:

```
int main(int argc, char *argv[])
{
    int  *pi     = &argc;

    printf("There are %d command line arguments.\n", *pi);
    return 0;
}
```

In this example the string, `pi`, is the string (identifier) in our C source file that identifies an object in memory (variable) that is big enough to hold the address of an integer (value).

So `pi` is an identifier for a chunk of memory big enough to hold one *address*, nothing more.

We assign a value to `pi`. That means that we say to the compiler, "Copy the address of the beginning of the memory chunk identified by the string, `argc`, into the memory chunk identified by the string, `pi`."

After the contents have been copied, the value of the object identified by `pi` is meaningful, namely, it's the address of the same object identified by `argc`. That means that we can access the value of argc by using the address that is stored in `pi`.

So in the call to printf, `*pi` means, "Look at the region of memory identified by `pi` to find a value that is a valid memory address. Then tell me the value in the object located at that address." *Two* values are being used here: the value that is stored in the pointer (it's a memory address), and the value stored in the object whose address is stored in the pointer.

Looking up one value by using a pointer value like that is called *indirection*.

4 friend compiler

The compiler is your friend. When you are programming, it will go through your code non–judgementally but thoroughly, pointing out things that don't make sense. If you ask it to, the compiler will also find parts of your code that look questionable.

When you turn on all the compiler warnings, the compiler will find uninitialized variables, unused variables — all kinds of simple mistakes that humans are wont to make. It may seem easier to leave off the compiler warnings and use a sloppy coding style, but in the end, that is the more difficult way!

The easy way is to get used to writing in a clean coding style with all compiler warnings turned on. That way you can write with abandon, compiling often, possibly using make (**see page 4**) to speed things up).

An interesting and productive way to think of C is as a language for two audiences: compilers and humanity. You have responsibilities to each audience. If you keep the compiler in mind when you write code, it will be very easy to remember things like casts and function prototypes, since you know that the compiler needs to know about information like types in order to create correct object code for you.

If you keep humans in mind, then you will be a happy human when you revisit your old code in the future! You will also write programs with fewer bugs, since easy-to-understand code makes for fewer bugs.

5 make

In order to befriend the compiler, you need to like using it *a lot*, so that you develop a relationship where it is doing more of your work. The more often you compile, the faster you can go, since the compiler will catch mistakes while they are fresh in your mind.[3] But that's only true if you have a convenient way of running the compiler and using its feedback.

The *make* utility is designed to help programmers to handle complex projects, but I find myself using it for trivial purposes, since it means that I can do everything in the build process for any given project by simply issuing the command, "make".

[3] And of course, you can go fast, because you have spent lots of time in design before beginning to write any code!

Many smart text editors, like emacs, have special features that allow the editor to interact with make and other programming tools. Emacs can capture the output of the compiler in a special window, and with keystrokes you can jump to the lines in the source that any compiler warnings or errors refer to.

Learning to use make means that you never have to be tied to any particular IDE or programming tool, since make can tie any commandline-driven tools together. Also, in a very large project or a well structured multi-file small project, make will speed up build time by compiling only those files that you've edited since the last build.

6 documentation

I am often dismayed at how many computer science students leave "Intro to UNIX" without knowing how to access online documentation for the tools that they use. In UNIX-like platforms, two major systems of documentation are *man pages* and *info* documentation.

The former is the traditional format for UNIX-like platforms. The command `man awk` should show you the manual for `awk`. (AWK is a neat tool.) Info is the documentation format of the Free Software Foundation developers. It's online version is hyperlinked and can be accessed with the command, `info`.

Lately, HTML format documentation is often placed in a location such as `/usr/doc`.

If you have trouble locating documentation, you can always get the source for the tool in question and read the README file. It should give you some pointers.