

C Programming, Part 3: Common Gotchas

Alex Kizer edited this page on Feb 15 · 5 revisions

What common mistakes do C programmers make?

Memory mistakes

String constants are constant

```
char array[] = "Hi!"; // array contains a mutable copy
strcpy(array, "OK");

char *ptr = "Can't change me"; // ptr points to some immutable memory
strcpy(ptr, "Will not work");
```

String literals are character arrays stored in static memory, which is immutable. Two string literals may share the same space in memory. An example follows:

```
char * str1 = "Brandon Chong is the best TA";
char * str2 = "Brandon Chong is the best TA";
```

The strings pointed to by `str1` and `str2` may actually reside in the same location in memory.

Char arrays, however, put that string literal in static memory, *and then copy it over to stack memory*. These following char arrays do not reside in the same place in memory.

```
char * arr1[] = "Brandon Chong didn't write this";
char * arr2[] = "Brandon Chong didn't write this";
```

Buffer overflow/ underflow

```
#define N (10)
int i = N, array[N];
for( ; i >= 0; i--) array[i] = i;
```

C does not check that pointers are valid. The above example writes into `array[10]` which is outside the array bounds. This can cause memory corruption because that memory location is probably being used for something else. In practice, this can be harder to spot because the overflow/underflow may occur in a library call e.g.

Edit

New Page

▼ Pages 51

Home

#Example Markdown

#Informal Glossary

#Piazza: When And How to Ask For Help

C Programming, Part 1: Introduction

C Programming, Part 2: Text Input And Output

C Programming, Part 3: Common Gotchas

C Programming, Part 4: Debugging

Deadlock, Part 1: Resource Allocation Graph

Deadlock, Part 2: Deadlock Conditions

File System, Part 1: Introduction

File System, Part 2: Files are inodes (everything else is just data...)


File System, Part 3: Permissions

File System, Part 4: Working with directories

File System, Part 5: Virtual file systems

Show 36 more pages...

Clone this wiki locally

 Clone in Desktop

```
gets(array); // Let's hope the input is shorter than my array!
```

Returning pointers to automatic variables

```
int *f() {
    int result = 42;
    static int imok;
    return &imok; // OK - static variables are not on the stack
    return &result; // Not OK
}
```

Automatic variables are bound to stack memory only for the lifetime of the function. After the function returns it is an error to continue to use the memory.

Insufficient memory allocation

```
struct User {
    char name[100];
};
typedef struct User user_t;

user_t *user = (user_t *) malloc(sizeof(user));
```

In the above example, we needed to allocate enough bytes for the struct. Instead we allocated enough bytes to hold a pointer. Once we start using the user pointer we will corrupt memory. Correct code is show bellow.

```
struct User {
    char name[100];
};
typedef struct User user_t;

user_t * user = (user_t *) malloc(sizeof(user_t));
```

Using uninitialized variables

```
int myfunction() {
    int x;
    int y = x + 2;
    ...
}
```

Automatic variables hold garbage (whatever bit pattern happened to be in memory). It is an error to assume that it will always be initialized to zero.

Assuming Uninitialized memory will be zeroed

```
void myfunct() {
    char array[10];
    ...
}
```

```
char *p = malloc(10);
```

Automatic (temporary variables) are not automatically initialized to zero. Heap allocations using malloc are not automatically initialized to zero.

Double-free

```
char *p = malloc(10);
free(p);
// .. later ...
free(p);
```

It is an error to free the same block of memory twice.

Dangling pointers

```
char *p = malloc(10);
strcpy(p, "Hello");
free(p);
// .. later ...
strcpy(p, "World");
```

Pointers to freed memory should not be used. A defensive programming practice is to set pointers to null as soon as the memory is freed.

It is a good idea to turn free into the following snippet that automatically sets the freed variable to null right after:(vim - ultisnips)

```
snippet free "free(something)" b
free(${1});
${1} = NULL;
${2}
endsnippet
```

Logic and Program flow mistakes

Forgetting break

```
int flag = 1; // Will print all three lines.
switch(flag) {
    case 1: printf("I'm printed\n");
    case 2: printf("Me too\n");
    case 3: printf("Me three\n");
}
```

Case statements without a break will just continue onto the code of the next case statement. Correct code is show bellow. The break for the last statements is unnecessary because there are no more cases to be executed after the last one. However if more are

added, it can cause some bugs.

```
int flag = 1; // Will print all three lines.
switch(flag) {
    case 1:
        printf("I'm printed\n");
        break;
    case 2:
        printf("Me too\n");
        break;
    case 3:
        printf("Me three\n");
        break; //unnecessary
}
```

Equal vs equality

```
int answer = 3; // Will print out the answer.
if (answer = 42) { printf("I've solved the answer! It's %d", answer);}
```

Undeclared or incorrectly prototyped functions

```
time_t start = time();
```

The system function 'time' actually takes a parameter (a pointer to some memory that can receive the time_t structure). The compiler did not catch this error because the programmer did not provide a valid function prototype by including `time.h`

Extra Semicolons

```
for(int i = 0; i < 5; i++) ; printf("I'm printed once");
while(x < 10); x++ ; // X is never incremented
```

However, the following code is perfectly OK.

```
for(int i = 0; i < 5; i++){
    printf("%d\n", i);;;;;;;;;;;;;;
}
```

It is OK to have this kind of code, because the C language uses semicolons (;) to separate statements. If there is no statement in between semicolons, then there is nothing to do and the compiler moves on to the next statement

Other Gotchas

C Preprocessor macros and side-effects

```
#define min(a,b) ((a)<(b) ? (a) : (b))
int x = 4;
if(min(x++, 100)) printf("%d is six", x);
```

Macros are simple text substitution so the above example expands to `x++ < 100 ? x++ : 100` (parenthesis omitted for clarity)

C Preprocessor macros and precedence

```
#define min(a,b) a<b ? a : b
int x = 99;
int r = 10 + min(99, 100); // r is 100!
```

Macros are simple text substitution so the above example expands to `10 + 99 < 100 ? 99 : 100`

Assignments in Conditions

```
int a = 0;
if (a = 1) {
    printf("What is a?\n");
}
```

Notice the second line-- `a = 1` vs. `a == 1` . What happens here? The assignment operator in C returns the value on the right. So in this case, `if (a = 1)` evaluates to `if (1)` .

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

