C Programming, Part 1: Introduction

Cornel Punga edited this page on Mar 20 · 7 revisions

Want a quick introduction to C?

- Keep reading for the quick crash-course to C Programming below
- Then see the C Gotchas wiki page.
- And learn about text I/O.
- Kick back relax with Lawrence's intro videos
- ^ and the same link includes a virtual machine-in-a-browser you can play with
- Or watch some old CS241 slides CS241 Old Slides

External resources

- C for C++/Java Programmers
- C Tutorial by Brian Kernighan
- c faq
- C Bootcamp
- Add your favorite resources here

Crash course intro to C

Warning new page Please fix typos and formatting mistakes for me and add useful links too.*

How do you write a complete hello world program in C?

```
#include <stdio.h>
int main(void) {
    printf("Hello World\n");
    return 0;
}
```

Why do we use '`#include stdio.h`'?

We're lazy! We don't want to declare the `printf` function. It's already done for us inside the file '`stdio.h`'. The #include includes the text of the file as part of our file to be compiled.

How are C strings represented?

As characters in memory. The end of the string includes a NULL (0) byte. So "ABC" requires four(4) bytes. The only way to find out the length of a C string is to keep reading memory until you find the NULL byte. C characters are always exactly one byte each.

When you write a string literal `"ABC"` in an expression the string literal evaluates to a char pointer (char *), which points to the first byte/char of the string. This means `ptr` in the example below will hold the memory address of the first character in the string.

```
char *ptr = "ABC"
```

How do you declare a pointer?

A pointer refers to a memory address. The type of the pointer is useful - it tells the compiler how many bytes need to be read/written.

```
int *ptr1;
char *ptr2;
```

How do you use a pointer to read/write some memory?

if 'p' is a pointer then use "*p" to write to the memory address(es) pointed to by p.

```
*ptr = 0; // Writes some memory.
```

The number of bytes written depends on the pointer type.

What is pointer arithmetic?

You can add an integer to a pointer. However the pointer type is used to determine how much to increment the pointer. For char pointers this is trivial because characters are always one byte:

```
char *ptr = "Hello"; // ptr holds the memory location of 'H'
ptr += 2; //ptr now points to the first'l'
```

If an int is 4 bytes then ptr+1 points to 4 bytes after whatever ptr is pointing at.

```
char *ptr = "ABCDEFGH";
int *bna = (int *) ptr;
bna +=1; // Would cause iterate by one integer space (i.e 4 bytes on some systems
ptr = (char *) bna;
printf("%s", ptr);
/* Notice how only 'EFGH' is printed. Why is that? Well as mentioned above, when
return 0;
```

Because pointer arithmetic in C is always automatically scaled by the size of the type that is pointed to, you can't perform pointer arithmetic on void pointers.

What is a void pointer?

A pointer without a type (very similar to a void variable). You can think of this as a raw pointer, or just a memory address. You cannot directly read or write to it because the void type does not have a size.

This is often used when either a datatype you're dealing with is unknown or when you're interfacing C code with other programming languages.

Does `printf` call write or does write call `printf`?

`printf` calls `write`. `printf` includes an internal buffer so, to increase performance `printf` may not call `write` everytime you call `printf`. `printf` is a C library function. `write` is a system call and as we know system calls are expensive. On the other hand `printf` uses a buffer which suits our needs better at that point

How do you print out pointer values? integers? strings?

Use format specifiers "%p" for pointers, "%d" for integers and "%s" for Strings. A full list of all of the format specifiers is found here

Example of integer:

```
int num1 = 10;
printf("%d", num1); //prints num1
```

Example of integer pointer:

```
int *ptr = (int *) malloc(sizeof(int));
*ptr = 10;
printf("%p\n", ptr); //prints the address pointed to by the pointer
printf("%p\n", &ptr); /*prints the address of pointer -- extremely useful
when dealing with double pointers*/
printf("%d", *ptr); //prints the integer content of ptr
return 0;
```

Example of string:

```
char *str = (char *) malloc(256 * sizeof(char));
strcpy(str, "Hello there!");
printf("%p\n", str); // print the address in the heap
printf("%s", str);
return 0;
```

Strings as Pointers & Arrays @ BU

How would you make standard out be saved to a file?

Simplest way: run your program and use shell redirection e.g.

```
./program > output.txt

#To read the contents of the file,
cat output.txt
```

More complicated way: close(1) and then use open to re-open the file descriptor. See http://angrave.github.io/sysassets/web/chapter1.html

What's the difference between a pointer and an array? Give an example of something you can do with one but not the other.

```
char ary[] = "Hello";
char *ptr = "Hello";
```

Example

The array name points to the first byte of the array. Both `ary` and `ptr` can be printed out:

```
char ary[] = "Hello";
char *ptr = "Hello";
// Print out address and contents
printf("%p : %s\n", ary, ary);
printf("%p : %s\n", ptr, ptr);
```

The array is mutable, so we can change its contents (be careful not to write bytes beyond the end of the array though). Fortunately 'World' is no longer than 'Hello"

In this case, the char pointer `ptr` points to some read only memory (where the statically allocated string literal is stored), so we cannot change those contents.

```
strcpy(ary, "World"); // OK
strcpy(ptr, "World"); // NOT OK - Segmentation fault (crashes)
```

We can, however, unlike the array, we change `ptr` to point to another piece of memory,

```
ptr = "World"; // OK!
ptr = ary; // OK!
ary = (..anything..) ; // WONT COMPILE
// ary is doomed to always refer to the original array.
printf("%p : %s\n", ptr, ptr);
strcpy(ptr, "World"); // OK because now ptr is pointing to mutable memory (the ar
```

What to take away from this is that pointers * can point to any type of memory while C arrays [] can only point to memory on the stack. In a more common case, pointers will point to heap memory in which case the memory referred to by the pointer CAN be modified.

`sizeof()` returns the number of bytes. So using above code, what is sizeof(ary) and sizeof(ptr)?

`sizeof(ary)`: `ary` is an array. Returns the number of bytes required for the entire array (5 chars + zero byte = 6 bytes) `sizeof(ptr)`: Same as sizeof(char *). Returns the number bytes required for a pointer (e.g. 4 or 8 for a 32 bit or 64 bit machine)

Which of the following code is incorrect or correct and why?

```
int* f1(int *p) {
    *p = 42;
    return p;
} // This code is correct;
```

```
char* f2() {
    char p[] = "Hello";
    return p;
} // Incorrect!
```

Explanation: An array p is created on the stack for the correct size to hold H,e,l,l,o, and a null byte i.e. (6) bytes. This array is stored on the stack and is invalid after we return from f2.

```
char* f3() {
    char *p = "Hello";
    return p;
} // OK
```

Explanation: p is a pointer. It holds the address of the string constant. The string constant is unchanged and valid even after f3 returns.

```
char* f4() {
    static char p[] = "Hello";
    return p;
} // OK
```

Explanation: The array is static meaning it exists for the lifetime of the process (static variables are not on the heap or the stack).

How do you look up information C library calls and system calls?

Use the man pages. Note the man pages are organized into sections. Section 2 = System calls. Section 3 = C libraries. Web: Google "man7 open" shell: man -S2 open or man -S3 printf

How do you allocate memory on the heap?

Use malloc. There's also realloc and calloc. Typically used with cast and a sizeof. e.g. enough space to hold 10 integers

```
int *space = (int *) malloc(sizeof(int) * 10);
```

What's wrong with this string copy code?

```
void mystrcpy(char*dest, char* src) {
  // void means no return value
  while( *src ) { dest = src; src ++; dest++; }
}
```

In the above code it simply changes the dest pointer to point to source string. Also the nul bytes is not copied. Here's a better version -

```
  while( *src ) { *dest = *src; src ++; dest++; }
  *dest = *src;
```

Note it's also usual to see the following kind of implementation, which does everything inside the expression test, including copying the nul byte.

```
  while( (*dest++ = *src++ )) {};
```

How do you write a strcpy replacement?

```
// Use strlen+1 to find the zero byte...
char* mystrdup(char*source) {
   char *p = (char *) malloc ( strlen(source)+1 );
   strcpy(p,source);
   return p;
}
```

How do you unallocate memory on the heap?

Use free!

```
int *n = (int *) malloc(sizeof(int));
*n = 10;
//Do some work
free(n);
```

What is double free error? How can you avoid? What is a dangling pointer? How do you avoid?

A double free error is when you accidentally attempt to free the same allocation twice.

```
int *p = malloc(sizeof(int));
free(p);

*p = 123; // Oops! - Dangling pointer! Writing to memory we don't own anymore

free(p); // Oops! - Double free!
```

The fix is firstly to write correct programs! Secondly, it's good programming hygiene to reset pointers once the memory has been freed. This ensures the pointer cant be used incorrectly without the program crashing.

Fix:

```
p = NULL; // Now you can't use this pointer by mistake
```

What is an example of buffer overflow?

Famous example: Heart Bleed (performed a memcpy into a buffer that was of insufficient size). Simple example: implement a strcpy and forget to add one to strlen, when determining the size of the memory required.

What is 'typedef' and how do you use it?

Declares an alias for a types. Often used with structs to reduce the visual clutter of having to write 'struct' as part of the type.

```
typedef float real;
real gravity = 10;
// Also typedef gives us an abstraction over the underlying type used.
// For example in the future we only need to change this typedef if we
// wanted our physics library to use doubles instead of floats.

typedef struct link link_t;
//With structs, include the keyword 'struct' as part of the original types
```