A Brief C Primer for CS 350

Benjamin Cassell

University of Waterloo
becassel@uwaterloo.ca

September 8, 2017

Overview

Introduction

Welcome to the CS 350 C Primer tutorial!

Why will we be using C for assignments in CS 350?

- C is flexible and gives the programmer total control.
- C is light-weight at its core, making it simple to use and easy to understand (usually).
- C gives you control over underlying memory management.
- And most importantly of all...

Why C?

(a) Using C to code.

(b) Using other languages to code.

A scientific and unbiased comparison of C with other languages.

Data Types

Data types in C define several things:

- The size of data.
- How data is arranged in memory.
- The legal operations that can be performed on data.

This course uses a 32-bit version of MIPS:

- `signed`/`unsigned` `char` = 1 byte
- `signed`/`unsigned` `short` = 2 bytes
- `signed`/`unsigned` `int` = 4 bytes
- `signed`/`unsigned` `long` = 4 bytes
- `signed`/`unsigned` `long long` = 8 bytes
- Pointers of any type = 4 bytes

Data Types

Warning!

New versions of C provide `bool` (through `stdbool.h`) and OS/161 provides something similar. This does not exist everywhere. In C, non-zero values are `true`, and zero values are `false`.

Warning!

Data sizes vary by system, with only loose guarantees on minimum type sizes, and which sizes are larger than others. When in doubt, use the `sizeof` keyword.

Warning!

`float` and `double` are omitted, as they do not work in OS-level code. Using `float` or `double` in OS/161 will crash the kernel.

Structs

Structs are special types in C:

- Structs are comprised of other data types called members.
- The size of a struct depends on the size of its member types.
- Members of a struct are accessed using the '.' operator.

Structs

```c
struct foo {
    int x;
    char a, b, c, d;
};

int main(int argc, char** argv) {
    struct foo bar;
    bar.x = 1; // Assigning a value to a struct's member
    bar.a = 2;
    printf("%d\n", (int)sizeof(bar));
    return 0;
}
```

This sample prints "8" and terminates.

Structs

Structs are *usually* laid out according to the way they are defined. In the previous example:

- The first 4 bytes are reserved for `int x`.
- The remaining 4 bytes are given to the four `char` variables.

Warning!

Your struct's size will **NOT** always be the summed total of its members' sizes! Compilers often pad structs and align values to memory addresses. It is always good practice to use the `sizeof` keyword instead of relying on your own guesswork!

Structs

```
struct foo {
    char x;
    int y;
};

int main(int argc, char** argv) {
    struct foo bar;
    printf("%d\n", (int)sizeof(bar));
    return 0;
}
```

This sample will likely print “8” and terminate.

Structs

It's kind of annoying to have to type `struct` every time you wish to declare one. The `typedef` keyword allows you to shortcut this by declaring a custom type:

```
typedef struct {
    int x, y;
} foo;

int main(int argc, char** argv) {
    foo bar;
    return 0;
}
```

Enums

Often, programmers wish to categorize things into specific types. The `enum` type provided by C helps give this capability:

- Enumerated types are simply dressed-up integer values.
- Enumerated types provide both compiler safety and optimization for operations involving items with different categories.
- An `enum` in C is often used in conjunction with a `switch` statement. These statements are often compiler-optimized into jump tables when many values are present, and can sometimes be more efficient than using `if-else` statements.

Enums

```
enum fruit {
    BANANA,
    ORANGE
};

void eat_fruit(enum fruit my_fruit) {
    switch (my_fruit) {
        case BANANA:
        case ORANGE:
            printf("Yum, healthy!\n"); break;
        default:
            printf("Item was probably a donut.\n"); break;
    }
    return;
}
```

Enums

Enums can be given explicit values if desired:

```
enum colours {
    RED = 1,
    ORANGE = 2,
    YELLOW = 4,
    GREEN = 8,
    BLUE = 16,
    INDIGO = 32,
    VIOLET = 64
};
```

Enums

Like structs, enums can be made easier to use with `typedef`:

```
typedef enum {
    DOG,
    SHARK
} animal;

void check_animal(animal my_animal) {
    if (my_animal == DOG) {
        pet(my_animal);
    } else {
        run_from(my_animal);
    }
    return;
}
```

Pointers

Pointers are C's method of allowing the programmer to directly manipulate memory:

- Pointers are variables that contain a memory address.
- The data contained at the address is inferred by the pointer type:
  - `int*` int_ptr: A pointer to an `int`.
  - foo* foo_ptr: A pointer to a foo.
  - `void*` ptr: A generic pointer with no particular type.
  - `int**` int_ptr_ptr: A pointer to a pointer to an `int`.

Pointers

Warning!

The position of the * in a pointer declaration is irrelevant, but you may start a minor war based on your decision. Although foo* my_foo, foo * my_foo and foo *my_foo are all equivalent, the suggested style (and the one used by the Linux kernel) is foo *my_foo. Just be sure to stay consistent!

Warning!

Be **VERY** careful when declaring multiple pointers on the same line. The * in a declaration only applies to the particular variable it is attached to.

Pointers

A bad declaration of multiple `int*`'s:

```
int* foo, bar, foobar;
```

Properly declaring multiple `int*`'s:

```
int* foo, * bar, * foobar;
```

Pointers

Pointers can be assigned in a variety of ways:

- Grabbing the address of a variable with the '&' operator:

```
int foo = 1;
int* bar = &foo;
```

- Casting other numerical values:

```
int* foo = (int*)0xFFFF0000;
unsigned int bar = 0xFFFF0000;
int* foobar = (int*)bar;
```

- Using a predefined macro like `NULL` (which is equal to 0):

```
int* foo = NULL;
```

Pointers

Remember, a pointer contains an address of a value, not the value itself. We can examine the address itself if desired:

```
printf("0x%x\n", foo); // Will print "0xFFFF0000"
```

To examine the value that resides at the address a pointer points to, use the dereference operator '*':

```
printf("%d\n", *bar); // Will print "1"
```

Pointers

Pointers are particularly useful for "passing by reference":

- Parameters are normally passed by value.
- Passing a pointer allows the programmer to manipulate the original variable from within a function.

```
void foo(int* bar) {
    *bar = *bar + 1;
}

int main(int argc, char** argv) {
    int bar = 0;
    foo(&bar);
    printf("%d\n", bar); // Will print "1"
    return 0;
}
```

Pointers

Pointers are very useful for passing structs around:

- Structs are passed by value by default.
- Passing by reference is more efficient and allows you to modify a struct from anywhere!

```
foo my_foo;
foo* foo_ptr = &my_foo;
do_something(&my_foo);
```

- When accessing the members of a struct via a pointer, the '.' operator annoyingly takes precedence over the '*' operator:

```
*foo_ptr.x = 0; // Wrong!
(*foo_ptr).x = 0; // Right!
```

Pointers

- To avoid issues with operator precedence and members of structs passed by reference, C has the '->' operator.

```
(*foo_ptr).x = 0; // Right!
foo_ptr->x = 0; // Also right!
```

- Remember, pointers can be made for any type of memory, including enums and other pointers!
- The `void` pointer is a pointer that represents a memory containing untyped data. Often `void` pointers are used to work with multiple different types of data or memory buffers.
- Because `char*` points to memory divided into bytes, it too is often used to work with memory buffers.

Arrays

Arrays are collections of elements of a single type:

```
int foo[32];
```

- This example creates an array of 32 integers.
- The total size occupied by an array is `sizeof` (type) * num_elements.
- Elements are accessed using the '[]' operator.
- The elements of an array are laid out contiguously in memory.
  - If the address of foo[0] is 0x10, then the address of foo[1] would be 0x14.

Arrays may be of any type, including structs and enums:

```
struct foo bar[32];
```

Arrays

In C, arrays are managed with pointer arithmetic:

- The variable name of the array is, in reality, just the memory address of the first element of the array (in other words, &foo[0] is equal to foo).
- Using the '[]' operator actually does the following:

```
int foo[32];
int bar1 = foo[8];
int bar2 = *(foo + 8); // Identical to bar1
int bar3 = *((int*)(((unsigned int)foo) +
    sizeof(foo[0]) * 8)); // Identical to bar1
```

- This is why arrays are zero-based in C.

Arrays

Warning!

An common pitfall arises from forgetting that pointers use pointer arithmetic. Also common is improper casting when manipulating pointers. Use explicit casting and brackets when in doubt.

Warning!

Pointers and arrays are similar, but **NOT** identical. Declaring a pointer allocates memory to store an address. Declaring an array only allocates for the elements, meaning the array variable itself cannot be mutated. Using '&' on a pointer results in a pointer to a pointer. Using '&' on an array simply results in the same value as the array variable!

Arrays

The similarities between pointers and arrays, and C's deference of memory management responsibility to the programmer enables some cool tricks:

```
typedef struct {
    int x;
    int y;
} foo;

int main(int argc, char** argv) {
    char bar[sizeof(foo) * 32]; // bar is an array of char
    foo* foobar = (foo*)bar; // Treat bar as an array of foo
    int* barfoo = (int*)bar; // Treat bar as an array of int
    return 0;
}
```

Arrays

Warning!

Calling `sizeof` on an array declared inside the same function as the call returns the total byte size of all elements in the array. Calling `sizeof` on a pointer to an array only returns the size of a pointer. Functions that accept arrays as arguments in reality accept a pointer to the array, and calling `sizeof` on the "array" in that function will only return the size of a pointer. It almost always more useful to write functions that accept pointers instead of arrays, as it more directly represents reality.

Warning!

In general, it is more useful to call `sizeof` on a variable's name than its type. That way, if the type of the variable changes, you do not need to modify the call to `sizeof`.

Arrays

Typed "dynamic arrays" are also provided as data structures in OS/161 (through some macro-based black magic):

- Great for storing data if you don't know how many items you'll need to accommodate up front (i.e. no reasonable upper bound).
- ...And that's about it.

Warning!

Don't use a dynamic array when a regular one would suffice. If you know how many items you need to store, or you have a reasonable upper bound, use a C array. This will improve your performance, and greatly reduce the strain on your sanity.

Strings

In C, strings are simply arrays of type char:

- Strings end with the NULL character, '\0' (equal to 0).
- A string can be allocated locally on the stack:

```
char foo[] = "Hello World!";
// {'H','e','l','l','o',' ','W','o','r','l','d','!','\0'}
// foo[1] == 'e', foo[12] == '\0'
printf("%c\n", *foo); // Prints out "H"
```

- Strings can also be declared as constants (or literals):

```
char* foo = "Hello World!";
```

- Literals are stored in the bss section of the executable file.

Strings

Remember that a pointer can be directed to point at any memory address at all, making something like the following possible:

```
char foo[] = "Hello World!";
char* bar = foo;
// foo[i] and bar[i] are now the same for any i
// sizeof(foo) and sizeof(bar) are still different
```

You can even declare strings to be larger than their contents:

```
char foo[] = "Hello World!";
char bar[13] = "Hello World!"; // foo and bar are identical
char foobar[20] = "Hello World!"; // Has 7 uninitialized bytes
```

Strings

Don't forget to properly treat C strings like arrays of characters:

```
int bad_comparison(const char* foo) {
    return (foo == "This is a dumb thing to do.");
}
```

The above comparison will NOT do the right thing. Use a library call like `strncmp` instead.

Warning!

NEVER use an unbounded string function like `strcmp` when a bounded version such as `strncmp` exists. Unless you like introducing security exploits for hackers to abuse. If so, go ahead.

Volatile Variables

When compiling a program in C, the compiler performs a variety of tricks to optimize your code:

- This can include re-writing code, eliminating variables, and even removing certain code altogether!
- Declaring a variable as `volatile` tells the compiler that the variable can change from outside of the context of the code.
  - Perhaps by another program or thread.
  - Perhaps by memory-mapped hardware.
- `volatile` can prevent unintended optimizations on a variable.

Volatile Variables

A compiler can change something like this...

```
int foo() {
    int bar = 0;
    for (i = 0; i < 10; i += 1) {
        bar += i;
    }
    return bar;
}
```

...into this:

```
int foo() {
    return 45;
}
```

Volatile Variables

Making a variable `volatile` usually stops the optimization:

```
int foo() {
    volatile int bar = 0;
    for (i = 0; i < 10; i += 1) {
        bar += i;
    }
    return bar;
}
```

Volatile Variables

Here is a very common real-world example:

```
unsigned int foo; // foo is memory-mapped
void wait_foo() {
    foo = 0;
    while (foo != 255);
}
```

This could be optimized into the following:

```
unsigned int foo; // foo is memory-mapped
void wait_foo() {
    foo = 0;
    while (1);
}
```

Volatile Variables

`volatile` fixes the issue:

```
volatile unsigned int foo; // foo is memory-mapped
void wait_foo() {
    foo = 0;
    while (foo != 255);
}
```

File Manipulation

C allows for direct interaction with and manipulation of files:

- People are often intimidated by file manipulation in C, but once you're used to using it, it's just as easy as file manipulation in any other language.
- Open files in C are dealt with through file descriptors, numbers which represent open files and allow you to interact with them.
- Your main tools when interacting with files will be the `open`, `close`, `read` and `write` system calls.
- All of these calls are very well-documented online.
- If you don't close open files you can run out of file descriptors!

Writing Files

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char** argv) {
    int i, rc, fd;
    unsigned int buf[40];
    for (i = 0; i < 40; i+= 1) {
        buf[i] = i;
    }
    fd = open("test-output", O_CREAT | O_WRONLY, S_IRWXU);
```

Writing Files

```
    if (fd < 0) {
        printf("Couldn't open file!\n");
        exit(1);
    }

    rc = write(fd, buf, sizeof(buf));

    if (rc < 0) {
        printf("Couldn't write to file!\n");
        exit(1); // Should really close fd before doing this
    }

    close(fd); // Should check the return code of this
    exit(0);
}
```

Reading Files

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

#define PER_ROW 4

int main(int argc, char** argv) {
    int i, rc, fd;
    unsigned int buf[40];
    if ((fd = open("test-output", O_RDONLY)) < 0) {
        exit(1);
    }
```

Reading Files

```
    if ((rc = read(fd, buf, sizeof(buf))) < 0) {
        exit(1); // Should really perform cleanup here
    }
    for (i = 0; i < 40; i += 1) { // Should use rc and sizeof
        if (i % PER_ROW == 0) {
            printf("offset = %4d : ", i * sizeof(unsigned int));
        }
        printf("0x%08x ", buf[i]);
        if ((i + 1) % PER_ROW == 0) {
            printf("\n");
        }
    }
    close(fd);
    exit(0);
}
```

Reading Files

```
offset = 0 : 0x00000000 0x00000001 0x00000002 0x00000003
offset = 16 : 0x00000004 0x00000005 0x00000006 0x00000007
offset = 32 : 0x00000008 0x00000009 0x0000000A 0x0000000B
offset = 48 : 0x0000000C 0x0000000D 0x0000000E 0x0000000F
offset = 64 : 0x00000010 0x00000011 0x00000012 0x00000013
offset = 80 : 0x00000014 0x00000015 0x00000016 0x00000017
offset = 96 : 0x00000018 0x00000019 0x0000001A 0x0000001B
offset = 112 : 0x0000001C 0x0000001D 0x0000001E 0x0000001F
offset = 128 : 0x00000020 0x00000021 0x00000022 0x00000023
offset = 144 : 0x00000024 0x00000025 0x00000026 0x00000027
```

Memory

Programs typically touch three distinct areas of memory:

- The stack, consisting of local variables determined at compile time.
- The heap, consisting of dynamic memory allocated at run-time.
- Global data, consisting of values loaded from the binary.

Stack

The stack is one of the most basic sections of memory:

- Both local variables and function arguments are allocated on the stack.
- The stack grows downwards on MIPS and most other systems.
- Stack spaces are handed out by the OS to threads.
- As functions are called, new stack frames are created by moving the stack pointer downwards.
- As functions return, the stack is unwound by reversing the stack pointer.
- Room created on the stack for variables is uninitialized by default (even for structs and arrays).

Stack

Every variable in the following example is located on the stack:

```
void foo (int bar) {
    int foobar;
    return;
}
```

Warning!

It is entirely possible to run out of space on the stack. By default, most OS's will give you "only" a few megabytes of stack space!

Stack

Warning!

One of the most common and basic mistakes you can make is to reference uninitialized variables, structs and array elements (e.g. using '+=' with an uninitialized integer, or assuming pointers are initialized to `NULL`). Get in the habit of initializing everything, and zero out entire structures or arrays if you have to!

Warning!

It is a critical mistake to reference stack variables from outside of their context. Passing variables downwards to functions is fine, but don't try to pass pointers to parameters or stack variables upwards!

Stack

Don't ever do this:

```
int* foo1 (int* bar) {
    int foobar = *bar;
    return &foobar; // This is almost certainly wrong!
}

int* foo2() {
    int bar = 0;
    int* foobar;
    foobar = foo1(&bar); // Passing bar downwards is fine
    return foobar; // Normally fine, but wrong because of foo1
}
```

Heap

Sometimes we can't know at compile time exactly how much memory we need. This is where dynamic memory comes in handy:

- Dynamic memory is issued from an area called the heap which is managed by the operating system.
- The heap is typically much larger than the stack by default.
- Storage is requested from the heap using the `malloc` system call which accepts a size (in bytes) and returns a pointer to a block of heap memory.
- Unlike the stack, no automatic unwinding ever occurs for the heap. When you're finished with heap memory it must be returned to the OS using the `free` system call.

Heap

Warning!

Calling `malloc` can return `NULL` if no memory is available!

Warning!

Like the stack, heap memory is uninitialized. Don't forget to initialize the memory your heap-based pointers point to!

Warning!

If you forget to return heap memory to the OS (for example in error cases), you will encounter memory leaks! This forgotten memory won't be available for use until the program terminates!

Heap

Sample heap usage:

```
int* foo() {
    int* bar = (int*)malloc(sizeof(int));
    *bar = 1;
    return bar;
}

int main(int argc, char** argv) {
    int* bar;
    bar = foo();
    printf("%d\n", *bar); // Will print out "1"
    free(bar); // May be a good idea to set bar = NULL after
    return 0;
}
```

Heap

Warning!

Once you've called `free` on a block of memory, there's no telling what it points to. It is now a **dangling** pointer. You should be extremely careful not to use this pointer again until it is reassigned to another value. Good practice may even be to manually set it to `NULL` when you are done with it!

Warning!

Calling `free` on the same pointer twice is dangerous! Although `free` on a `NULL` pointer does nothing at all, calling `free` on a bad memory address or one that has already had `free` called on it causes undefined behaviour. Undefined behaviour is bad, precisely 101 times out of 100.

Heap

Using free in good and bad ways:

```
void good(int** foo) {
    free(*foo);
    *foo = NULL; // Kind, brave people code like this
    return;
}

void bad(int** foo) {
    free(*foo);
    free(*foo); // Fools and charlatans code like this
    bar(*foo); // Convicted war criminals code like this
    return;
}
```

Endianness

Endianness is an important consideration when working with C on various platforms:

- Endianness defines byte orders in memory.
- Intel's x86 architecture uses little endian semantics.
- Little endian means least significant bytes come first.
- System/161 uses big endian semantics.
- Big endian means most significant bytes come first, and is what you are logically used to.

Little endianness is a legacy of older processors, as it was slightly more efficient for some mathematical operations and also requires fewer instructions for casting values.

Endianness

Here are what bytes look like in little and big endian:

```
unsigned int x = 0xDEADBEEF;

Little endian: Least significant byte at lowest address
0 .. 7 8 .. 15 16 .. 23 24 .. 31
[ EF ] [ BE  ] [  AD  ] [  DE  ]

Big endian: Most significant byte at lowest address
0 .. 7 8 .. 15 16 .. 23 24 .. 31
[ DE ] [ AD  ] [  BE  ] [  EF  ]
```

Endianness

Here is what the sample from file reading looks like when dumped on a little endian system:

```
offset = 0 : 0x00000000 0x01000000 0x02000000 0x03000000
offset = 16 : 0x04000000 0x05000000 0x06000000 0x07000000
offset = 32 : 0x08000000 0x09000000 0x0A000000 0x0B000000
offset = 48 : 0x0C000000 0x0D000000 0x0E000000 0x0F000000
offset = 64 : 0x10000000 0x11000000 0x12000000 0x13000000
offset = 80 : 0x14000000 0x15000000 0x16000000 0x17000000
offset = 96 : 0x18000000 0x19000000 0x1A000000 0x1B000000
offset = 112 : 0x1C000000 0x1D000000 0x1E000000 0x1F000000
offset = 128 : 0x20000000 0x21000000 0x22000000 0x23000000
offset = 144 : 0x24000000 0x25000000 0x26000000 0x27000000
```

The "Evil" Goto Command

goto is controversial and has a bad reputation in C:

- All goto does is transfer control to a labelled statement.
- And yet, there are academic essays with titles such as *Go To Statement Considered Harmful*, *"GOTO Considered Harmful" Considered Harmful*, and of course, *"'GOTO Considered Harmful" Considered Harmful' Considered Harmful*.

The following code prints out "world":

```
int main(int argc, char** argv) {
    goto my_first_label;
    printf("hello");
my_first_label:
    printf("world");
    return 0;
}
```

Proper Use of Goto

goto is actually pretty common in large code bases (including operating systems), and when used responsibly can reduce code duplication and unnecessary flag variables. For example:

```
void long_loops(void) {
    for (i = 0; i < 100000; i += 1) {
        for (j = 0; j < 100000; j += 1) {
            if (some_condition(i, j)) {
                goto exit_loops;
            }
        }
    }
exit_loops:
    // Run some other code
}
```

Proper Use of Goto

Consider this code that allocates and deallocates objects:

```
char* buffer1 = NULL, * buffer2 = NULL, * buffer3 = NULL;
if ((buffer1 = malloc(4096)) == NULL) {
    return ERROR;
}
if ((buffer2 = malloc(4096)) == NULL) {
    free(buffer1);
    return ERROR;
}
if ((buffer3 = malloc(4096)) == NULL) {
    free(buffer2);
    free(buffer1);
    return ERROR;
}
return SUCCESS;
```

What would this code look like using goto?

Proper Use of Goto

```
char* buffer1 = NULL, * buffer2 = NULL, * buffer3 = NULL;
if ((buffer1 = malloc(4096)) == NULL) {
    goto cleanup;
}
if ((buffer2 = malloc(4096)) == NULL) {
    goto cleanup;
}
if ((buffer3 = malloc(4096)) == NULL) {
    goto cleanup;
}
return SUCCESS;
cleanup:
if (buffer1 != NULL) {
    free(buffer1);
}
// ...Check and free other non-NULL buffers
return ERROR;
```

Improper Use of Goto

Don't use `goto` for literally anything else. Really.

- Especially don't `goto` backwards.
- Especially don't `goto` into nested statements.
- Especially don't `goto` past important variable initializations.
- Especially don't `setjmp` and `longjmp` (unless you really know what you're doing).

General Tips

A few generic tips for programming success:

- Useful tools: GDB is non-optional. Cscope and Ctags are both useful. Valgrind is great outside of CS 350.
- Learn to use the C pre-processor. It is fairly powerful.
- Initialize all your variables.
- Make sure you understand pointers inside and out.
- Check the return code of all system calls that return a value.
- Modularize, comment, and write consistent code.
- Compile and test frequently. Automate testing, if possible.
- Use source control. Like GDB, this is non-optional.

Useful Resources

- The Linux man pages.
  - System calls are in chapter 2 (e.g. man 2 `open`).
  - Library calls are in chapter 3 (e.g. man 3 `strncpy`).
  - Searchable and hyperlinked man pages: `linux.die.net/`
- Wikipedia.
- Various "pitfalls of C" and "expert C tricks" pages.
  - Bit Twiddling Hacks: `graphics.stanford.edu/~seander/bithacks.html`
  - Beej's Guide to Network Programming: `beej.us/guide/bgnet/`

The End

Congratulations! You're on your way to becoming an expert C programmer! Everyone will want to use your OS!

Questions?