C Review for CS 246 Students

May 13, 2020

Before We Begin...

This is not a extensive review about C; this is just the subset of what you have learned in CS 136. For an extensive guide to C please look for external resources.

Compiling in the command line

Use the gcc command.

```
gcc <filenames> -o <executable_name>
```

Basic syntax

```
// if statement
if (condition1){
    stmts
} else if (condition2){
    stmts
}
...
else{
    stmts
}
```

Basic syntax

```
// Loops
while(condition){
    ... // loop body
}
for (initialization; condition; update){
    ...
}
```

`break` exits the current loop.
`continue` skips the rest of current iteration and starts the next iteration.

Declaration, Definition and Initialization

Declaration Asserts enough about the existence of an entity to permit type-checking to proceed; no further details.

Definition Provides full details about the entity, and causes space to be set aside for it (in the case of variables and functions).

Initialization The variable being defined is given a starting value.

Declaration, Definition and Initialization

```
// Declaration
void foo(int x, int y);
extern int n;
struct Stack;

// Definition
void foo(int x, int y){return;}
int n;
struct Stack{
    int *arr;
    int start,end;
}; // Don't forget the semicolon

// Initialization
int n = 2; // This is the initialization syntax.

// Assignment
n = 4;
```

Pointers, arrays

```
// ip is a pointer to an int
int *ip;
// some more pointer definition.
// p1,p2,p3 are pointer points to an integer, i is an
    integer.
int *p1, *p2, *p3, i;
// vp is a pointer that could point to any type.
// Since the pointed object's size is unknown,
// the pointer cannot be dereferenced.
void *vp;
```

Pointer variable stores a memory address. At compile time, the only verification preformed on pointers is that they type check; it is possible to make pointer points to arbitrary locations of memory.

Pointers, arrays

```
// ia is an array of 4 integers.
int ia[4];
// ia2 is an array of 3 integers; the elements are
    already given.
int ia2[] = {1,1,2};
```

Note that although the name of the array is the shorthand for a pointer to the array's first element, **arrays and pointers are not the same.** However, when an array is passed as a function parameter, a pointer to the array's first item is what is actually passed.

Pointers, arrays

Accessing elements of an array can be performed by indexing and using pointer arithmetic.

```
int ia[] = {1,4,7};

ia[1]; // returns 4

*(ia + 2); // returns 7
```

Note: indexing is syntactic sugar for pointer arithmetic (meaning it does the same operation but the code easier to read).

Constants

The const keyword applies to the type specifier on it's immediate left; if const is the leftmost keyword, it applies on the type specifier on it's immediate right.

```
const int n = 2; // n is a constant integer

struct Posn{ int x,y; };
const struct Posn pos; // Cannot change Posn's fields

const int *ip = &n;
*ip = 10; // ERROR!
ip = &m; // OK

int * const ip2 = &n;
*ip2 = 10; // OK
ip2 = &m; // ERROR!

int const * const ip3 = &n; // Error on both assignment
    and dereferanced assignment
```

Strings

In C, strings are just char arrays.

```
// s1 and s2 are equivalent.
char s1[] = "Hello";
char s2[] = {'H','e','l','l','o','\0'};
// s3 have the same content as s1 and s2 but different
    sizes:
char s3[12] = "Hello";
// s4 points to an array of constant chars (in read-only
    memory); cannot change
// content of s4.
char *s4 = "Hello";
// Don't forget the fact that C string are just array of
    characters:
int bad_comp(const char *str){
    return (str == "This is a dumb thing to do.");
}
```

Structs

Structs are special types in C:

- Structs are consisted of other data types called members.
- the size of a struct depends on size of its member types (but is not always the sum of all it's member's sizes)
- Members of a struct are accessed using the '.' operator. Use '->' operator for the members of a pointer to the struct.

CS136 Memory Model

This the the memory model you have seen in CS136:

Code
Read-Only Data
Global Data
Heap
Stack

Scope

The scope of the variable is the lifetime of the variable.
In C, the scope of a variable is defined by the block of code it resides in. By block, we mean a section of code within brace brackets.
Some common blocks are functions, if statements and loops. Each iteration of a loop is a different scope. The scope of parameters to functions is the entire body of the function.
Global scope means the variable was defined outside of a function or structure and can be accessed for the lifetime of the program.

Stack Allocation

All the variables allocated on the stack are deleted at the end of their scope.

```
// this function does not do what you intend to do..
int *foo(int *x, int *y){
    int z = *x + *y;
    return &z; // VERY BAD
}
```

In this case, the memory used by z is already out of scope when foo returns. So this function will always return an invalid memory address.

Heap Allocation

```
#include<stdlib.h>
// an integer on the heap
int *p = malloc(sizeof(int));
// an array of n Posn
struct Posn *paposn = malloc(n * sizeof(struct Posn));
// ....
// freeing the memory
free(p);
free(paposn);
// The memory address pointed by p and paposn are now
    invalid (dangling pointer)
p = NULL;
paposn = NULL;
```

Heap Allocation

Some notes:

- Heap memory is uninitialized. Make sure you initialize it before working with heap memory.
- Calling `free` on the same pointer twice is very dangerous (and is undefined behaviour).

The code looks somewhat complicated. We will see a more elegant alternative in C++. We will also see a better alternative for `NULL` as well.

Common Memory Errors

Segmentation Fault (segfault): Accessing invalid memory locations. The offending memory address is often `NULL`.

```
int *p = NULL;
*p = 2;
```

Invalid read/write: Trying to access/write to memory locations that is not allocated.

```
int *p = malloc(2 * sizeof(int));
p[2] = 0; // invalid write
```

Memory Leak: Not freeing memory allocated on heap.

```
malloc(sizeof(char)); // a memory leak of one byte
```

Command Line Arguments

```
int main(int argc, char *argv[]) {
    ...
}
```

- `argc` is the number of command line arguments (at least 1)
- `argv` is an array of string, each one is one command-line argument
- `argv[0]` is the name of the program itself (as invoked in command line)
- `argv[1]`, `...`, `argv[argc-1]` are the actual arguments;
- `argv[argc]` is guaranteed to be a null pointer.