C Lectures

2.Variables

Salih MSA

May 15, 2022

Abstract

By the end of this presentation, you would've understood the following:

- Different datatypes in C
- The how of variable usage
- The why of variable usage

Primitive types

Integer types - 1

In C exists the following basic integer types:

- **(signed) char** - 1 byte in size. used typically to store characters of values [-127, +127]
- unsigned char - same size as above. stores characters of values [0, +255]
- (signed) short (int) - 2 bytes in size and is basically a smaller version of a normal Int. range: [-32,767, +32,767]
- unsigned short (int) - same size as above. range: [0, 65,535]
- **(signed) int** - 4 bytes in size. probably one of the most commonly use types. range: [-32,767, +32,767]
- **unsigned int** - same size as above. range: [0, 65,535]
- (signed) long (int) - 4 bytes in size. range: [-2,147,483,647, +2,147,483,647]
- unsigned long (int) - same size as above. range: [0, 4,294,967,295]

Datatypes

Integer types - 2

- (signed) long long (int) - 8 bytes in size. range: [-2,147,483,647, +2,147,483,647]
- unsigned long long (int) - same size as above. range: [0, 4,294,967,295]

Types in bold are most commonly used. You should ideally stick to these unless there's a specific need to use otherwise.

Datatypes

Floating-point types

In C exists the following basic floating-point types:

- **float** - (typically) 4 bytes [1]. range: [1.17549e-38, 3.40282e+38]
- **double** - (typically) 8 bytes [2]. range: [2.22507e-308, 1.79769e+308]
- long double - size varies between systems [3]

Types in bold are most commonly used. You should ideally stick to these unless there's a specific need to use otherwise.

---

[1] Following IEEE 754 single-precision binary floating-point format convention

[2] Following IEEE 754 double-precision binary floating-point format convention

[3] Following IEEE 754 quadruple-precision binary floating-point format convention: 16 bytes. x86 format: 10 bytes (but typically 12/16 bits in memory due to padding), or same as double in other cases

The how of variable usage

Variable use - 1

In C, we need to manually declare variables with a statement PRIOR to using them, using the following structure:

data-type variable-name ;

We can then use the variable, using the name we gave it, to alter the variable's value henceforth. For example:

```
1  int main(void)
2  {
3      int number ; /* declare */
4      number = 5 ; /* use */
5      return 0 ; /* end of program */
6  }
```

The how of variable usage

Variable use - 2

In the example above, what I did was **declare** the variable, then **initialise/define** it's value afterwards. However, this can be done on the same line (moreover, I recommend doing it on the same line).

```
1  int main(void)
2  {
3      int number = 5 ; /* declare and initialise */
4      return 0 ; /* end of program */
5  }
```

The how of variable usage

Theory - Restrictions

There are (sane) restrictions to variable use:

- Variables can only be **declared** where independent statements are valid. For example, they cannot be on the right hand of an equals symbol or on either side of a boolean expression [4](logically because how can you assign a declaration - it makes no sense)
- Variable identifiers are restricted to the scope they were defined in (see next slide).

[4]When we come to conditional statements, you will understand boolean expressions. What I mean by that is stuff like 'x is equal / not equal to y', 'x is larger / smaller than y', etc.

The how of variable usage

Scope - 1

A scope in programming is a region of the program where a defined variable exists - once we have exited it fully, it is no longer accessible. Scopes in C are defined by what is in between of a pair of curly braces (i.e. {}). Take the following code example:

```
1  int bigger_scope ; /* global variable */
2
3  int main(void)
4  {
5      int big_scope ; /* declare variable in main scope */
6      { /* create a scope (indicated by {}) */
7          int little_scope ; /* variable in local scope */
8          little_scope = 2 ;
9          big_scope = 3 ;
10     }
11     bigger_scope = 5 ;
12 }
```

The how of variable usage

Scope - 2

In the example above:

- Global scope [5]- variables declared here can be accessed from anywhere.
- Main function's primary scope - variables declared in the main function are only accessible in the main function's scopes and has access to all the scopes it is in (itself and the global scope)
- Main function's inner/local scope - variables declared in the local scope of the main function are only accessible in the local scope and has access to all the scopes it is in (itself, the main primary scope and the global scope)

---

[5] GLOBAL VARIABLES (variables defined in the global scope) are evil. Such reckless access to data makes code harder to debug and it's 100% availability makes it more error-prone. I was showing it purely for demonstrative examples. See `https://www.tutorialspoint.com/why-are-global-variables-bad-in-c-cplusplus`

The how of variable usage

Your turn!

Copy ‘my_first_program.c’ from the previous lecture and call it ‘variable_use.c’. In it’s main function, create the following variables **exactly as requested**:

- An integer type called x. DECLARE IT ONLY.
- A float called y. Declare and initialise the variable on the same line with the value 5.1
- A char called m. Declare and initialise the variable on the same line with your favourite letter

Hint: I have not told you HOW to assign to characters yet. Figure it out.

The why of variable usage

Theory - 1

Previously explained has been how to make use of variables. However, I feel I should explain WHY C makes you declare variables as you do (i.e. why it makes you state the variable, why mention the type, why scope it) - it comes down to how memory is generally allocated and used by C programs:

1. There is an indicator used to internally track the start & end of a local stack frame (i.e. a scope [6])
2. When a scope is created, the start indicator jumps to the current position of the end indicator [7]

[6] Here's the thing: it is much more typical that NAMED scopes (i.e. functions) go through this whole process, as opposed to these inner scopes. But the compiler can TECHNICALLY do whatever it needs to do to produce a code which performs well - this can be framing everything to framing nothing. There's also stack frame related stuff done to conform to the host system's Application Binary Interface. I guess what I am trying to say a bunch of these steps (all except 3 on the next page) are implementation specific. But it does happen and it is bogstandard assembly logic, so listen please. Love, Salih x

[7] See footnote 6

The why of variable usage

Theory - 2

3. For each variable added, we write x amount of bytes from the last memory entry, where x is the SIZE of the variable's datatype
4. When it comes to the end of the scope, the end indicator hops back to the position of the start indicator and the start indicator moves back to where it was before, effectively making all the variables in the now-exited scope ready to be overwritten again [8]

[8] See footnote 6

The why of variable usage

Example - Program - 1

Here is an example C function with comments [9]:

```
1  int main(void)
2  {
3      int x = 2 ; /* allocate 4 byte var: end indc - 4 */
4      return 0 ;
5  }
```

.

[9] Note the stack size *increases* backwards, hence why it shows, when adding variables, the end tracker is made to point to smaller addresses

The why of variable usage

Example - Program - 2

The below is an example of corresponding yet simple assembly code [10]:

```
1  global    main
2  section   .text
3  main:
4          push         rbp ; store old base
5          mov         rbp, rsp ; set base as end
6          sub rsp, 4  ; minus 4 bytes from end
7          mov         WORD [rsp], 2 ; mov value 2 to variable at end
8          mov rsp, rbp ; set value of end as old value (held currently by
9          pop         rbp ; set base back to old value
10         mov eax, 0 ; set return code to 0 (ignore)
11         ret ; exit function
```

[10] I know this is a C tutorial BUT I felt that this was important to show you why scoping works as it does - these are an example of what the compiler might add behind the scenes. If you want to run this assembly: 'nasm -felf64 bye_asm.asm gcc bye_asm.o ./a.out', where GCC is used for final linkage because I wanted code to fit on the slide