COMP101P

Principles Of Programming

Taught by Graham Roberts and David Clark

Notes compiled by students

Contents

Chapter 1

C Programming

C is the grandad of most of the languages we use today. It's so good that people still use it to create the glorious Linux operating system. So, let's C what it can do!

Additional Notes
CHammond made some decent notes on most of this stuff if you want a quick but thorough overview of the module. Check them out on Google Drive if you have access, here: `https://docs.google.com/document/d/1t2VUYKYipdOwf7I22OtLSfnoMIe7KPTwmSjbXqMY5FI/edit`

1.1 Basics

C is an *imperative programming* language, which means that you write programs by saying "do this!" then "do that!". Things get a little more complex than that when you're dealing with bigger programs (Linux kernel, anyone?) but essentially that's what it's about. Here's a quick rundown of things you can do.

1.1.1 Variables

Variables are like, super important. They are the bread and butter of imperative programming, they are ways of storing values so that you can use them again or manipulate them however you desire. Let's look at a few different variable types and how you can define them. I'll also point out here that comments (notes you can add to the code that have no effect on the running) look like `/* this */` or like `// this`.

```c
// Integers (whole numbers)
int someNumber = 42; // yer standard integer, 16 bits in size.
short yolo = 256; // a smaller integer.
char hiya = 25; // a very small integer, range of -127 to +127 makes it
    awesome for text characters.
long bigNum = 5484198; // For when you want to store biiiig numbers.

// Floating point numbers (numbers with decimal points)
float someNum = 3.4;  // single-precision, 32-bits.
double preciseNum = 3.232365;  // double-precision (durr)

bool amIRetakingTheYear = true;  // true or false, you need stdbool.h
   though.
```

Hey look at that, you can write C! To define a variable, you add the type you want in front of the name you're going to give it. You can then, if you want, assign it a value with the assignment (=) operator, or just leave it. To then access that variable or its value again, you only need its name.

1.1.2 Simple Operations

Operations are built-in mathsy things you can perform on variables. Here's a few you can do:

Assignment

Sets the value of a variable.

```
    lecturesMissed = 5;
```

Addition

Adds two numbers together. You can also just add a number on by combining this with the assingment operator.

```
    someStupidExample = 4 + 8; // is now 12.
    someStupidExample += 5;  // someStupidExample = 12 + 5 = 17.
```

Subtraction

Works **exactly the same** as addition, just with a minus (-) sign instead of a plus sign. Yes, this is starting to bore me now.

Increment and Decrement

You can add or subtract a single value with these. You have the choice of applying them before (prefix) or after (suffix). Check this:

```
    int some_num = 10;
    some_num++;  // some_num is now 11
    ++some_num;  // some_num is now 12
    some_num--;  // some_num is now 11
```

But why do you get the two separate ways of incrementing or decrementing numbers? It's because, when used in an expression, the prefix operator will use the new value in the expression, whereas the postfix will not. Let's see an example because my explanation is kinda bad.

```c
int i = 10;
int k = ++i; // i = 10 + 1 = 11, THEN k = i = 11;
int k = i++; // k = i = 11, THEN i = 11 + 1 = 12;
```

This is the cause of both misery and joy in my opinion. Use it at your own risk.

Multiplication

Again, works just like addition but with an asterisk (*).

```c
imBored = 5 * 12;  // is now 60.
```

Division

Hey, guess what? It works in EXACTLY THE SAME WAY. Honestly, I should be getting money for this. But what I will point out is that dividing an integer by an integer returns the whole part of a result, not rounded.

```c
int x = 5;
int y = 2;
x = x / y;  // x is now 2.
```

Modulo

It's very possible to get the remainder of dividing integers in C, you must use the modulo or remainder operator.

```c
ex = 5 % 2;  // 1, 5/2 is 2.5 or 2 with 1 left over
```

Pointer (De)reference

Woah there friend, you may need to slow down!

1.1.3 Order Of Operations

Pretty much 100% confirmed as the first question in the exam. Like in real world maths, operators in C follow a 'precedence', e.g. some are always performed before others. This is known as order of operations, or operator precedence. It goes like this:

- **1 (first)** - postfix increment/decrement, brackets etc.
- **2** - prefix increment/decrement, pointer stuff
- **3** - multiplication, division, modulo
- **4** - addition, subtraction

- ... - a bunch of stuff that may not be so important right now.

Thanks, `http://en.cppreference.com/w/c/language/operator_precedence`.

1.1.4 Arrays

An array is a list of variables for which each have its own index. Basically, when you make your array, you get a big chunk of memory where you can store n number of values... so naturally you can't change its size since that would be a shambles. But rather than try to explain memory voodoo to you, I'll leave that for later and just show you an example of how arrays work.

You define an array similarly to variables, except you're also providing a size.

```c
int someList[30];

// let's give a list values!
int anotherList[5] = {1, 3, 5, 6, 8};
```

It's probably best to think of arrays as like pigeon holes, where the number on the box is how you can refer to it.

Figure 1.1: A pigeon hole. By "Stacalusa" on Wikipedia.

You can reference an item in the array just like any other variable, you just type `myArray[x]`, where `x` represents the index of the item (arrays count from 0 though, so beware!).

Since you can have arrays of any type (as long as each item has the same type as each other), you can actually have arrays of arrays, or 2-d arrays. They are simply defined like so:

```c
int some_2d_array[3][3] = {
    {1, 2, 3},
    {4, 5, 6],
    {7, 8, 9}
};

some_2d_array[1][2] = 4;
```

Oh oops, maybe I should point out at the same time that text is represented in C using *strings*, which are quite literally an array of characters. Here's the difference though. You can define strings (value doesn't change), by using this funky pointer notation (it will be explained later): `char *my_string = "Dat boi";` They can also be represented with your standard ol' `char meme[10]` or whatever. Another difference is that strings have a hidden character at the end to say that the string is finished, called a 'null' character. It's represented as `\0` and literally serves no other purpose.

1.1.5 Call functions

C comes with a bunch of which are blocks of code which do things. Some functions take in values and some return values, which makes them really handy for not writing the same damn piece of code a million times over. Most of them require you to import the related code *library* to your program. You can do this like so, at the beginning of your file:

`include <stdlib.h>` - include a built-in library called "stdlib".

`include "mylib.h"` - include a library that is stored in the same directory which is called "mylib.h".

`include "libs/somelib.h"` - include a library that is stored in a directory called "libs".

There's a couple of neat ones that you won't go wrong with remembering. They may come back when you least expect it!

- **stdlib.h** - the 'standard' library which contains things like memory allocation, and value conversion (based `atoi()`, thank you).
- **stdio.h** - input/output library. Expect things like printing to console (`printf()`) as well as reading, along with file stuff too.
- **string.h** - helpful string library, which contains a bunch of helper functions such as comparing strings (`strcmp()`) and so on.
- **stdbool.h** - allows you to use the `bool` type, which is just true/false.
- **math.h** - this will save your butt, as well as containing mathsy things like trigonometry functions, you also get some neat constants like `M_PI` which gives you Pi. Awesome!

1.1.6 User-Defined Functions

So, you want to make your own functions, huh? Very well. Functions are defined with, in order: the return variable type, the function name, as well as its *parameters* (if any), followed by the content of the function. Let's look at an example, we'll call it matt, and it will take an integer, add 5, and return it.

```
int matt(int input) {
    input += 5;
    return input;
}
```

That wasn't too bad was it? Note the "return" keyword. What this does is that it returns whatever you put after it. In this case, it's returning the value of `input`. This doesn't have to be a variable; if you had a function called `give_me_lemon` for instance, you could always have `return "lemon";` Also note that for each parameter, you need to provide its type. Very important.

Every program in C must have a main function, which returns either an `int` or `void`. Void is a magic keyword that means, "this function does not return anything". The main function is the entry point, the way in, to your program. Set things up there, I recommend.

1.1.7 If Statements And Logic

Maybe there's some code you don't want to run all the time. If statements are your solution!

```c
if (logical_statement) {
    do_something();
}
else if (another_statement) {
    do_something_else();
}
else {
    give_up();
}
```

Glossary

F | I | L | S | V

F

function

A block of code that performs a certain task. It can also return values (or not), if the programmer desires. 5

I

imperative programming

Programming without any fancy classes or anything. Essentially, do this then do that. 2, 5

L

library

A selection of functions and/or constant values which you can share around or whatever. 5

S

string

How text is stored in programs. Normally using an array of characters. 5

V

variable

a stored value which is referred to by its name. 2, 5