COMP101P

Principles Of Programming

Taught by Graham Roberts and David Clark

Notes compiled by students

Contents

1 C Programming **2**
1.1 Basics . . . 2
1.1.1 Literals . . . 2
1.1.2 Variables . . . 2
1.1.3 Simple Operators . . . 3
1.1.4 Order Of Operations . . . 4
1.1.5 Arrays . . . 5
1.1.6 Call functions . . . 6
1.1.7 User-Defined Functions . . . 6
1.1.8 If Statements And Logic . . . 7
1.2 Intermediate Programming . . . 9
1.2.1 Loops . . . 9
1.2.2 Bait And Switch . . . 10
1.2.3 A Big Hand For Our Cast . . . 10
1.2.4 A Couple Of Pointers . . . 11
1.2.5 Rollin' In The Heap . . . 12
1.2.6 In N' Out . . . 13

Glossary **15**

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 Literals

Simply just the value. Using a literal is known as hard-coding as it cannot be changed when the program is running.

1.1.2 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`.

```
// 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.

1.1.3 Simple Operators

Operators 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.

```
    int i = 10;
2   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 (*).

```
1   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.

```
1   int x = 5;
    int y = 2;
3   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.

```
1   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!

1.1.4 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`.

1.1.5 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.

```
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:

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

```
6
    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.

1.1.6 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!

1.1.7 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.

```
1   int matt(int input) {
        input += 5;
3       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.

1.1.8 If Statements And Logic

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

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

It's pretty self-explanatory. If the logical statement holds true, then the code in the first block is used. If it is false, then the else statement is used. In this case above, the else statement contains another if-else statement, testing for another statement. So only if both logical statements are false, then the last code block will be used.

But what is a logical statement? It's an expression that returns true or false. For this, we can use *logical operators* to take variables and reduce them to something boolean. Let's explore them.

Equals Operator

The equals operator takes two variables, literals, functions or whatever of the same type and returns true if they are equal, and false if not. You type `==` to use it.

```
int my_num = 5;
if (my_num == 5) {
    // this will execute as my_num is equal to 5.
}
```

Don't use this for strings! You should use the `strcmp()` function from stdlib.h for that.

Not Equals Operator

Does the exact opposite of what equals does, funnily enough. This is represented with `!=`. Do you really need an example for this?

Less Than (Or Equal To) Operator

Given two values, you can test if one is less than the other. You can use `<` for just that, or you can be bold and use `<=` to additionally make the statement true if the two values are equal. Superb!

```
    int my_score = 10;
    if (my_score < 30) {
        give_up(); // this will be executed.
    }
```

Greater Than (Or Equal To) Operator

Uhm, it's the opposite of the less than operator.

Logical Not Operator

This will invert a logical statement or boolean value given to it, represented just by the `!` character.

```
    if (!true) {
        // this will NEVER run since !true becomes false.
        // Why you would write code like this, I don't know.
        // But if you ever do, I recommend you take a cold hard
        // look at your code and wonder just why.
    }
```

Logical Or Operator

This takes two logical statements and will return true if *either* of them are true. You use the `||` (two pipes) characters for this.

```
    int gimme_five() {
        return 5;
    }

    int x = 10;
    if ((x == 10) || (x == gimme_five())) {
        // x == 10 so this is true. Never mind that x is not equal
        // to the output of gimme_five()
    }
```

Logical And Operator

This returns true only if both statements are true. This is represented with the `&&` characters.

```
bool do_you_need_this(int score, bool failed) {
    if ( (score > 30) && !failed) {
        return false;
    }
    else {
        return true;
    }
}
```

So nice one, you can do that now!

1.2 Intermediate Programming

Oh, so you think you're a big guy now? You've finished the basics stuff and you want something more, something a little more complex? Well you've come to the right place buster, it's time for some intermediate C.

1.2.1 Loops

Loops are great because you can repeat stuff many times without having to write it out each time. You can repeat stuff without having to write it out each time, which is great.

While Loop

While loops are run infinitely while the logical statement is true. So you gotta be careful otherwise it can run forever if your logical statement doesn't change. Unless you want that to happen, you maverick you.

```
int i = 0;
while (i < 10) {
    printf("Hello");
}
// prints out HelloHelloHelloHelloHelloHelloHelloHelloHelloHello
```

If you are a true renegade, you can put the logical statement at the end to create a "do-while" loop. This is then checked at the *end* of each loop, which means that the loop is guaranteed to run at least once.

```
int i = 0;
do {
    printf("Hello");
} while(i > 1);
// prints out Hello
```

For Loop

Oh, now you've done it. So you're not content enough with just a while loop; you want something that looks more badass. So I give you the for loop. You have statement to run before the loop, a state to check at the beginning of each loop, and a statement to run at the end of each loop. It looks a bit like this.

```
for (inital_thing(); test_this(); at_the_end_of_each_loop();) {
    do_something();
}

// In practice:
int i;
for (i = 0; i < 10; i++) {
    printf("%d",i);
}
// Prints "12345678910"
```

You get me? You have me got? Good.

1.2.2 Bait And Switch

The switch statement is a kinda neat way of testing for many different values without having some massive if-else if-else if and so on. You specify a "switch", and then test for values, which **must** be literals, as well as a default. Each 'case' as they are known must end with a `break` so that the code exits the switch statement rather than go down to the next one.

```
char input = get_some_input();

switch (input) {
    case 'a': menu_option_1();
              break;
    case 'b': printf("2B or not 2B");
              break;
    default:  printf("Nice work, you mug.");
              break;
}
```

1.2.3 A Big Hand For Our Cast

Type casting is a handy way of converting certain variable types to another. It's useful for converting floats to integers, for example.

```
double x = 4.5;
int y = (int) x;
// y is now 4.
```

1.2.4 A Couple Of Pointers

So this is the one that makes people cry. And honestly, pointers shouldn't if you treat them right! A pointer is kinda what you'd imagine it to be, a type of variable that stores a memory address, so it literally "points" to an actual variable. When you create a variable normally, e.g. `int why = 10`, you're asking what's called the "heap" memory for some space to store an integer. It'll get the memory address of that space it's been allocated, so that the program can then write into that space.

What is a memory address you ask? Memory on a computer is divided into loads of little sections of a certain size, and they are numbered so that that space can be remembered. Think of each space like a car parking space, and the address is the number painted on the floor in front of the space, and the data is the car... I'm not sure where I'm going with this analogy.

The cool thing with C is, you can very much fiddle with the memory addresses to your liking (the memory given to your program by the operating system, that is). But as a wise Uncle Ben once said, with great power comes great responsibility. And you really do have the potential to mess up your program with pointers (although your computer is safe).

General pointer stuff is baked into the C language, however you may need to use the `stdlib.h` library to access some of the memory functions.

Get Thine Address

The `&` operator gets the address of a variable. No man, nothing to do with the logical AND operator! You put it in front of a variable name to return its address.

```
1    int x = 101;
     printf("%d\n", &x); // this prints out 0x4328db or some such thing.
```

Pointer Type

A pointer type is completely different to its variable bretheren. It just stores a pointer and nothing else, although the type of pointer determines just how big the memory space that it's pointing at is. A pointer type is defined using the asterisk, in the form `<type> *<var_name>`.

```
     int some_val = 346;
2    int *my_ptr = &some_val;
     // my_ptr now points at the location of some_val
```

Dereference the pointer

You have a pointer which is nice, but it'd be rather swish if you could get the value it's pointing at right? Hence you need the dereference operator. You have to use the asterisk before the variable's name, and bam there you go.

```
1    int some_val = 645;
     int *my_ptr = &some_val;
```

```
    some_val += 10;
    printf("my_ptr looks at %d\n", *my_ptr);
    // prints out "my_ptr looks at 655"
    // ARE YOU A MAGICIAN
```

The cool thing about this is that you can use pointers to pass in variables to functions rather than just their values, so you can change them in-situ. They also have their uses in arrays. Say whaaaat? You heard me, arrays use pointers already. The index format (`array[10]`) is just syntactic sugar for `*(array + 10)`, which means get the address of array, and add 10 to it, then return whatever's stored there (array items are stored in adjacent memory locations, fortunately). This is why you may see arrays passed into functions or what have you as something like `int *some_array` if the length isn't known, or how strings are passed as `char *my_string`.

Pass By Reference

Whenever you use a variable in a parameter for a function, you're giving the function the value of the variable rather than the variable itself. But you can change this if you wish, by passing the variable pointer instead. Let's look at this interesting example.

```
void add_five(int *num) {
    *num += 5;
}

int main() {
    int x = 10;
    add_five(&x); // address of x is passed into add_five since it
   needs a pointer.
    // The value of x is now 15.
}
```

Hey, that ain't too hard!

1.2.5 Rollin' In The Heap

So it's all well and good doing that stuff above with pointers, but you can also use the blighters to create and destroy variables in memory as you like. Introducing the heap, a section of memory that's dedicated *just* for this purpose.

Get Memory From The heap

```
    #include <stdlib.h>

    int *my_ptr = (int *) malloc (sizeof (int));
```

What you've got here, is that a new int pointer is created, which points at some new allocated memory. The `malloc()` literally means "Memory ALLOCate", and accepts a size as its parameter; in this case we're using the `sizeof()` function to get the size of an

int type. Finally, we use a type cast to make the pointer into an int pointer. But why stop there? Let's make an array using heap memory, since that's a thing right?

```
#include <stdlib.h>

int arr_size = 10;
float *array = (float *) malloc (arr_size * sizeof(float));

array[0] = 10.4; // first element
*(array + 2) = -23.653; // third element (counts from 0 right?)
```

Destroy Memory From The Heap

You're done with your data and you now want to release that piece of memory back to the heap to be used again. This is good because if you never release to the heap, you can run out of heap memory and you will cry.

```
#include <stdlib.h>

int * p = (int *) malloc( sizeof (int) );
*p = 10;
// blah blah blah
free(p);
// the address of p can now be overwritten
```

1.2.6 In N' Out

Programs are rather more handy when a user is able to interact with it. It does bring you some extra problems though (the user is an idiot), but that's well outside the scope of this course. Let's do some output. You'll need to include the `stdio.h` library for these functions.

printf

`printf()` lets you write out (or 'print' as the oldies call it) text to what's known as STDOUT. Normally this is just the screen, so let's go with that. You give printf a format string, followed by variables you want to output.

```
int x = 10;
printf("The pointless value of x is %d\n", x);
// prints out "The pointless value of x is 10"
```

Format Strings

Let's talk more about the format strings and why they look scary. You've got your standard text, as well as conversion characters and special characters. These are the common special characters:

- \ **n** - Newline character, use this to start a new line.
- \ **r** - Return character, if you're on Windows then this will probably be of use to you.
- \ **0** - Null character, this is at the end of every string.
- \ **t** - tab character, if you want to align your text all neat and that.

Want an actual backslash? Just use two backslashes next to each other.
Now, these are the conversion characters:

- % **s** - String.
- % **d** - Integer.
- % **f** - Floating point number.
- % **c** - Character.

You can also combine these with a number to represent a certain number of characters to display. So, `%4d` would be a nice way of displaying currency (probably). Want to use an actual percentage sign? Just use it twice again!

scanf

`scanf()` is one way of getting user input. It's the safest way of getting user text from

Glossary

F | I | L | O | P | S | V

F

function

A block of code that performs a certain task. It can also return values (or not), if the programmer desires. 11

I

imperative programming

Programming without any fancy classes or anything. Essentially, do this then do that. 2, 11

L

library

A selection of functions and/or constant values which you can share around or whatever. 6, 11

logical operator

An operator that returns true or false. 7, 11

O

operator

A symbol that represents a mathematical, relational or logical operation. `http://www.tutorialspoint.com/computer_programming/computer_programming_operators.htm`. 3, 11

P

parameter

An input for a function. 6, 11

S

string

How text is stored in programs. Normally using an array of characters. 6, 11

V

variable

a stored value which is referred to by its name. 2, 11