Lecture 4: Variables, Types, Printing to the Console

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/04-c-variables-types-printf.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

In this lecture we talk about variables, types, and printing to the console in C.

Variables

Here is an example of C code with a main function using a local **variable** named `a`:

```c
#include <stdio.h>

int main() {
    int a;                          // declare a of type int (signed integer)

    a = 12;                         // set the value of a to 12
    printf("a's value: %d\n", a);   // print the value of a

    return 0;
}
```

The variable is set to 12 and then printed to the standard output. Variables have a **name**, here it's `a`. Variables also have a **type**, here it's `int` corresponding to a signed integer. Finally, variables have a **value**, which can generally evolve as the program executes. Before being able to use a variable, it must first be declared. A variable **declaration** is a statement defining the name and type of the variable. Here we declare `a` in the first line of code of the `main` function. In C a variable name should start with a letter or underscore and contain a combination of letters, underscores, numbers. Try to be descriptive, for example here `sum` or `final_balance2` are much more meaningful than just `a`.

Using Variables

Here are a few examples of variables usage:

```c
int a; int b; int c;
int d = 12;  // declare and set
int x, y = 10, z = 11;

a = 12;      // set a to 12
b = 20;      // set b to 20

c = 10 + 10; // set c to 20
a = b;       // a = 20
```

```
d++;          // d = d + 1
y *= 2        // y = y * 2;
```

We first declare 3 integers `a`, `b` and `c`. A variable can be declared and set in one statement, this is what happens to `d`. On the third line we declare `x`, `y` and `z` and we also set the value of `y` to be 10 and the value of `z` to be 11. On the code on the right we have an example of arithmetic operation, setting the value of `c` to be `10+10`. We also set the value of `a` to be the value of `b`. And we also have some nice shortcuts for commonly-used operations. For example by using `d++` we effectively increment `d` by 1. Also, `y *=2` corresponds to multiplying `y` by 2 and storing the result in `y`. There is a similar construct for addition, subtraction and division.

Types

Each variable must have a **type**. Types are used by the compiler to do two things. First, allocate the memory space for the variable. Each type defines a given size in bytes for this. Second, the compiler performs some checks on the operations realised on the variable. For example in some cases it is able to warn about overflows when one tries to store a number larger than the size of the target variable type.

Types Define Data Storage Size in Memory

At runtime one can see the program's memory as a gigantic array of bytes. As an example, the `int` type, on the x86-64 architecture that most of our laptop/desktop computers are using, is 4 bytes long. So the compiler reserves 4 bytes in memory when declaring e.g. `int a`. That memory will be used to store `a`'s value at runtime.

Primitive Types

Beyond `int`, other basic types (named **primitive types**) include characters (`char`) and floating point numbers (`float` for single- and `double` for double-precision). We can describe a constant character in the code, here `x`, with single quotes. When performing an arithmetic operation on an integer and a `float`, the compiler will promote the result to a `float` (same for `double`). For example here `float_res` will include a decimal part:

```
int int1 = 2, int2 = 4;
float float1 = 2.8;
/* when mixed with floats in arithmetics, integers are promoted to floats */
float float_res = int1 + float1;
float float_res2 = int1 * (float1 + int2);
/* when stored in an integer variable, floats are _truncated_ */
int int_res = int1 * (float1 + int2);
```

Note the use of the parentheses to set the order of evaluation in the second example setting `float_res2`. Even if the right-hand side is promoted to a float in the assignment of `int_res`, because of its type (`int`), the result will be truncated.

More Types, Qualifiers

When declaring a variable, we use a combination of a type and optional qualifiers, that will define what the variable can hold and how much space is reserved. It is very important to be aware of the storage size of variables, as things like overflows can have very nasty consequences. Here are a few examples with the information that the C standard[1] gives about the corresponding storage sizes:

```
short int a;               // signed, at least 16 bits: [-32,767,         +32,767]
int b;                     // signed, at least 16 bits: [-32,767,         +32,767]
unsigned int c;            // unsigned:                 [0,               +65,535]
long int d;                // at least 32 bits:         [-2,147,483,647,  +2,147,483,647]
unsigned long int e;       // unsigned:                 [0,               +4,294,967,295]
long long int f;           // at least 64 bits:         [-9x10^18,        +9x10^18]
long long unsigned int g; // unsigned:                  [0,               +18x10^18]
float h;                   // storage size unspecified, generally 32 bits
double i;                  // storage size unspecified, generally 64 bits
```

[1] https://en.wikipedia.org/wiki/C_data_types

For example `int` is signed (i.e. it can hold positive as well as negative integers) and should be at least 2 bytes, so it can store numbers from -32,767 to 32,767. `unsigned int` has the same size and is unsigned, so it can store more, but only positive, numbers. Note that the storage size information coming from the standard are rather imprecise. The actual sizes depend on the architecture of the CPU the program is compiled to execute upon.

sizeof

To get the exact storage size we can use the `sizeof` function, that takes a type as parameter:

```
int so_short = sizeof(short int);
int so_int = sizeof(int);
int so_float = sizeof(float);
int so_double = sizeof(double);

printf("size of short:         %d bytes\n", so_short);
printf("size of int:           %d bytes\n", so_int);
printf("size of float:         %d bytes\n", so_float);
printf("size of double:        %d bytes\n", so_double);
```

On x86-64:

- The size of `short` is 2 bytes.
- The sizes `int`, `unsigned int` as well as `float` are 4 bytes.
- The sizes of `long`, `long long int`, and `double` are 8 bytes.

Printing to the Terminal

We have been using `printf` to print a lot of things to the console, let's see how it works more in details. It takes one or more arguments:

```
printf(<format string>, <variable1>, <variable2>, etc.);
```

The first argument is the format string containing the text to print as well as optional markers that will be replaced with variables' values The next arguments are optional. It is the list of variables which value needs to be printed, 1 variable per argument. The format string marker to use depends on the corresponding variable type. A few examples:

```
int int_var = -1;
unsigned int uint_var = 12;
long int lint_var = 10;
float float_var = 2.5;
double double_var = 2.5;
char char_var = 'a';
char string_var[] = "hello";

printf("Integer: %d\n", int_var);
printf("Unsigned integer: %u\n", uint_var);
printf("Long integer: %ld\n", lint_var);
printf("Float: %f\n", float_var);
printf("Double: %lf\n", double_var);
printf("Characters: %c\n", char_var);
printf("String: %s\n", string_var);
printf("Several variables: %d, %lf, %s\n",
    int_var, double_var, string_var);
```

Various markers are illustrated in that example:

- `%d` is used for signed integers, `%u` for unsigned ones.
- `%l` is used to indicate the `long` qualifiers, for example we have `%ld` for a `signed long int`.
- `%f` is used for floats and `%lf` for doubles.

- Finally, notice the declaration of a string with the brackets for the variable name and the double quotes for the string itself. We can print it with `%s`.