1 Very Short Introduction to C

Since it is my experience that long introductions do more evil than good when the audience has no idea about the subject matter, this introduction will be very brief, and to the point.

C is a programming language. It has been written by programers, and for programmers. It is considered to be a low-level language. C provides maximum flexibility, computing power, low-level control at the expense of hard-to-track bugs, long development times and hard-to-manage code.

In this course we will attempt to learn the syntax and basics of C, along with simple algorithmic and mathematical applications.

2 The C "Hello, world!" Program

It has been customary for some time to start programming languages with a program that will output the sentence "Hello, world!" on the screen. We will not break this rule. Here is the source code[1] for our "Hello, world!" program in C:

```
/* The "Hello, world!" program. */

#include <stdio.h>

int main(void)
{
    printf("Hello, world!");
}
```

If you type this program into the computer, compile it, and run it, it should print **Hello, world!** on the screen.

Here is a brief line-by-line explanation of the code:

- `/* The "Hello, world!" program.  */`

  This is a comment (comments are delimited using `/*  */` in C) and has no actual effect on the program itself.

- `#include <stdio.h>`

  This is what is called a *preprocessor directive*. What it does in effect is to insert the *standard header file* `stdio.h` at that position into the code. This file, `stdio.h` contains the definitions of standard input/output functions that are part of the *C standard library*. Here this is required in order to be able to use the `printf()` function in our program.

- `int main(void)`

  This is what is called a *function declaration*. All C programs consist of one or more functions. This line declares a function with the name `main`, which is a special function name; it is where the execution of a program will start.

[1] Don't worry if you do not know what "source code" and other terms we use here mean right now.

- {

  This denotes the start of the *function body*. All *code blocks* in C are delimited by a pair of braces.

- `printf("Hello, world!");`

  This is a `statement`. What the statement says is to call the `printf()` C library function with the argument ``` ``Hello, world!'' ```. In its simplest use, `printf()` prints its argument on the screen.

- }

  This denotes the end of the function body.

3 The C Language Syntax

The main parts of a C program are *functions*. Any C program consists of one or more functions. The general form of a function definition is given below:

```
<return-type> <function-name>(<argument list>)
{
    <function body>
}
```

In trying to explain what a function is, I will try to draw from your mathematical knowledge. A C function is similar to a mathematical function in that it takes a number of arguments, and produces a result. In mathematical functions, more often than not, the arguments and results are mathematical numbers. However, in C, both the arguments and the return value of a function can be a multitude of data types, so they must be given in the function definition.

So, if we go back to our `main()`, it reads:

```
int main(void)
```

The keyword `int` here means *integer*, and denotes an integer data type usually 4 bytes long. Why does `main()` return an integer? This might be a bit too deep to discuss now, but by the "new" C standards, the return value of `main()` *must* be `int`. It should be sufficient to say that the value will be returned to the underlying operating system when the program ends.

Now, you can see that where the argument list should be, we have but one word: `void`. This is a C keyword denoting a void data type. In other words, here it stands to say that the function `main()` takes no arguments at all.

At this point we know (more or less) what goes into and out of a function. However, we do not know what a function contains. A function consists of *statements*. A statement is basically an instruction to the computer to do something. It can be a function call (obviously, functions can and do call other functions), an assignment, a variable declaration and some other things. One key point is that all statements in C must end with a semicolon (';'). A newline character does not act as a delimiter in C. Extra whitespace (spaces, tab characters, newlines) are usually ignored by the C compiler. The only places you can not put extra spaces is in the middle of identifiers (variable names, function names) and C keywords.

In our first example program, the line:

```
printf("Hello, world!");
```

is a statement. It is a function call, calling the standard C library function `printf()` in order to print out a string.

One mystery remains before we can go on: How do we return values from a function? The C keyword `return`, followed by an expression is used to return a value from a function. Note that the use of `return` is twofold, first it determines what value will be returned to the calling function, and second it ends the execution of the function at that point. In other words, once a `return` statement is executed, any following statements within that function will not be executed, and execution will be returned to the calling function.

Now, we can add a `return` statement to our example program as below:

```
/* The "Hello, world!" program. */

#include <stdio.h>

int main(void)
{
    printf("Hello, world!");

    return 0;
}
```

This will make the program return the value 0 to the operating system, which means "everything worked just fine" by convention. Other return values from `main()` indicate that some sort of problem was encountered, again, by convention.

4 Variable Types in C

C is a strongly typed language, that is, all variables must be properly *declared* before they can be used. However, there is much room for confusion in variable types in C, since the C standard leaves a lot to the compiler implementation. In any case, we will try to cover the basic variable types now.

4.1 Integer Variable Types

There are four types of integer variables in C. These are `char`, `short int`, `int`, `long int`.

- `char`

  The `char` type is a special type, which is used to hold 8-bit characters. (As a side note, when you want to type character constants in your code, they are typed in single quotes, such as `'A'`.) Thus, the size of a `char` variable is one byte. However, you can use a variable of type `char` as a small integer as well.

- `short int`

  The `short int` type is typically shorter than an `int` and longer than a `char`. These days in common compilers `short ints` are 2 bytes long. Note that it is possible to use just `short` instead of `short int`.

- `int`

  The `int` type is the "main" integer type, and normally is as large as a register of the CPU it is running on. In the Intel architecture, `int`s are 4 bytes long.

- `long int`

  The `long int` type is supposed to be an integer type longer than the `int`. However, in my experience, in most compilers it is just as long as `int`, that is, 4 bytes long. Note that it is possible to use `long` instead of `long int`.

In addition to all these integer data types, there is also the matter of signed vs. unsigned integers. If you want to use negative numbers as well as positive numbers, you need to reserve one bit for the sign of the number. But, if you wish to denote only positive numbers, you can use the whole range for positive numbers. In C, by default, `short`, `int`, and `long` are signed. If you wish to use them as unsigned variables, you need to prepend `unsigned` to the variable type; for instance `unsigned int`. There also exists a `signed` keyword, but normally is not used for these types since it is the default.

However, the `char` is more problematic, its default behavior is left to the compiler. So, if you want to be sure of what you are doing (and will actually use a `char` as a small integer) you had better specify one of `signed` or `unsigned`.

So what are the actual ranges for these variables? Once you know the size of the variable in bytes, it is simple to figure out. Given the number of bytes of the variable is n the ranges of a signed and unsigned variable are as follows:

Type	Min	Max
Unsigned	0	$2^{8n} - 1$
Signed	-2^{8n-1}	$2^{8n-1} - 1$

So, for example, for a two-byte, signed `short int`, the range is -32768 to 32767.

4.2 Floating-Point Variable Types

Floating-point variables can hold numbers that have fractional values. There are two floating-point types in C, `float` and `double`. They can both hold rather large values, but `double` provides twice the precision that a `float` provides.

5 Declaring Variables

As mentioned earlier, since C is a strongly typed language, all variables must be properly declared before they can be used. The general form of a variable declaration is as follows:

```
<type> <variable-name>[= <initial-value>];
```

As you can see, it is optionally possible to assign an initial value to the variable being declared. So, for example, if we want to declare the `int` variable `i` with an initial value of 10, this would be our declaration:

```
int i = 10;
```

It is also possible to declare multiple variables of the same type in a single declaration as follows:

```
float x=1.0, y=2.0, z;
```

Note that here we have chosen *not* to initialize `z` to any value. The value of an uninitialized variable will be undefined (i.e., random) so it is not a good idea to use uninitialized variables without properly giving them a value first.

Where can declarations go? In C, declarations can only be present at the beginning of a block of code (a block of code is simply one or more statements enclosed in curly braces).

6 Another Example Program

Here is another example program that makes use of things we have seen so far:

```
/* Example two: Variables, functions, declarations... */

#include <stdio.h>

int multiply(int a, int b)
{
    int c;

    c = a*b;

    return c;
}

int main(void)
{
    int product;

    product = multiply(4, 5);
    printf("The product is: %d\n", product);

    return 0;
}
```

Here you can see variables, declarations and functions in action. There are four new things here. One is assignment:

```
c = a*b;
```

In an assignment, the right hand side gets calculated, and the result goes into the variable on the left. The second is also here, and it is multiplication. We will come back to operations soon enough, but for now, this is how two things are multiplied by each other.

The third and fourth things are in this line:

```
printf("The product is: %d\n", product);
```

Firstly this is a more "advanced" use of `printf()`. The first argument to `printf()` is the *format string*. The format string, in addition to normal characters, can contain a number of *directives*. Here, `%d` is a directive, saying "replace this with an integer". But what integer? That is given as the second argument to `printf()` – here, it is the variable `product`.

The last thing of note here is the "`\n`" within the string. It is what is called an "escape sequence", which corresponds to a newline character. By putting a newline character there, we make sure anything else printed by the program will go onto a new line, and not the same line as this one.