The structure of C programs

Let's look at the high-level structure of a short C program, rot.c (using ellipsis to omit some statements for now).

At this stage it's not important what the program is supposed to do.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* Compile this program with:
   cc -std=c99 -Wall -Werror -pedantic -o rot rot.c */
#define ROT 13
static char rotate(char c)
{
    ....
    return c;
}

int main(int argc, char *argv[])
{
    // check the number of arguments
    if(argc != 2) {
        ....
        exit(EXIT_FAILURE);
    } else {
        ....
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

Of note in this example:

- Characters such as a space, tab, or newline, may appear almost anywhere - they are stripped out and ignored by the C compiler.

  We use such whitespace characters to provide a layout to our programs. While the exact layout is not important, using a consistent layout is very good practice.

- Keywords, in **bold**, mean very specific things to the C compiler.

- Lines commencing with a '@' in **blue** are processed by a separate program, named the C *preprocessor*.

  In practice, our program is provided as input to the preprocessor, and the preprocessor's output is given to the C compiler.

- Lines in **green** are **comments**. They are ignored by the C compiler, and may contain (almost) any characters.

  C99 provides two types of comments -

  - /* block comments */ and
  - // comments to the end of a line
The structure of C programs, continued

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* Compile this program with:  
   cc -std=c99 -Wall -Werror -pedantic -o rot rot.c */
#define ROT 13
static char rotate(char c)
{   ..... 
    return c; 
}
int main(int argc, char *argv[])
{   // check the number of arguments 
    if(argc != 2) 
        {   ..... 
            exit(EXIT_FAILURE); 
        }
    else 
        {   ..... 
            exit(EXIT_SUCCESS); 
        }
    return 0;
}
```

Same program, but more to note:
- A variety of brackets are employed, in pairs, to group together items to be considered in the same way. Here:
  - angle brackets enclose a filename in a #include directive,
  - round brackets group items in arithmetic expressions and function calls,
  - square brackets enclose the index when access arrays (vectors and matrices...) of data, and
  - curly brackets group together sequences of one or more statements in C. We term a group of statements a block of statements.
- Functions in C, may be thought of as a block of statements to which we give a name. In our example, we have two functions - rotate() and main().
- When our programs are run by the operating system, the operating system always starts our program from main(). Thus, every complete C program requires a main() function.

The operating system passes some special information to our main() function, command-line arguments, and main() needs a special syntax to receive these.

When reading code aloud, we pronounce main()'s two parameters as "arg count" and "arg values".

- When our program finishes its execution, it returns some information to the operating system. Our example here exits by announcing either its failure or success.
Compiling and linking our C programs

C programs are human-readable text files, that we term source-code files.

This makes them very easy to copy, read, and edit on different computers and different operating systems.

C is often described as being portable at the source-code level.

Before we can run (execute) our C programs, we must translate, or compile, their source-code files to files that the operating system can better manage.

A program known as a compiler translates (compiles) source-code files into object-code files.

Finally, we translate or link one or more object-code files to produce an executable program, often termed a 'binary', an 'executable', or an 'exe' file.

A program known as a linker performs this translation, also linking our object-code file(s) with standard libraries and (optionally) 3rd-party libraries.

Depending on how we invoke the compiler, sometimes we can 'move' straight from the source-code files to the executable program, all in one step.

In reality the compiler is 'silently' executing the linker program for us, and then removing any unwanted object-files.
Variables
Variables are locations in a computer's memory. A typical desktop or laptop computer will have 4-16GB of memory, or four.... billion addressable memory locations.

A typical C program will use 4 bytes to hold a single integer value, or 8 bytes to hold a single floating-point value.

Any variable can only hold a single value at any time - they do not maintain a history of past values they once had.

Naming our variables
To make programs more readable, we provide variables with simple names. We should carefully choose names to reflect the role of the variable in our programs.

1. While variable names can be almost anything (but not the same as the keywords in C) there's a simple restriction on the permitted characters in a name -
   - they must commence with an alphabetic or the underscore character ( _ A-Z a-z), and
   - be followed by zero or more alphabetic, underscore or digit characters ( _ A-Z a-z 0-9).

2. C variable names are case sensitive, thus:

   MYLIMIT, mylimit, Mylimit and MyLimit

   are four different variable names.

3. Older C compilers may limit variable names to, say, 8 unique characters. Thus, for them,

   turn_reactor_coolant_on and turn_reactor_coolant_off

   are the same variable! Keep this in mind when writing portable code.

4. While not required, it's preferred that variable names do not consist entirely of uppercase characters.
   We'll consistently use uppercase-only names for constants provided by the C preprocessor, or user-defined type names:

   MAXLENGTH, AVATAR, BUFSIZE, and ROT
Basic datatypes

Variables are declared to be of a certain datatype, or just type.

We use different types to represent the permissible values that a program’s variable has.

For example, if we’re using a variable to just count things, we’ll use an integer variable to hold the count; if performing trigonometry on angles expressed in radians, we’ll use floating-point variables to hold values with both an integral and a fractional part.

C provides a number of standard, or base types to hold commonly required values, and later we’ll see how we can also define our own user-defined types to meet our needs.

Let’s look quickly at some of C’s base datatypes:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description, and an example of variable initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Boolean (truth values), which may only hold the values of either true or false</td>
</tr>
<tr>
<td></td>
<td>e.g. bool finished = false;</td>
</tr>
<tr>
<td>char</td>
<td>character values, to each hold a single values such as an alphabetic character, a digit character, a space, a tab...</td>
</tr>
<tr>
<td></td>
<td>e.g. char initial = ‘C’;</td>
</tr>
<tr>
<td>int</td>
<td>integer values, negative, positive, and zero</td>
</tr>
<tr>
<td></td>
<td>e.g. int year = 2006;</td>
</tr>
<tr>
<td>float</td>
<td>floating point values, with a typical precision of 10 decimal digits (on our lab machines)</td>
</tr>
<tr>
<td></td>
<td>e.g. float inflation = 4.1;</td>
</tr>
<tr>
<td>double</td>
<td>“bigger” floating point values, with a typical precision of 17 decimal digits (on our lab machines)</td>
</tr>
<tr>
<td></td>
<td>e.g. double pi = 3.1415926535897932;</td>
</tr>
</tbody>
</table>

Some textbooks will (too quickly) discuss the actual storage size of these basic types, and discuss the ranges of permissible values. When writing truly portable programs - that can execute consistently across different hardware architectures and operating systems - it’s important to be aware of, and avoid, their differences. We’ll examine this issue later, but for now we’ll focus on using these basic types in their most obvious ways.

From where does the bool datatype get its name? - the 19th century mathematician and philosopher, George Boole.
The scope of variables
The scope of a variable describes the range of lines in which the variable may be used. Some textbooks may also term this the visibility or lexical range of a variable.

C has only 2 primary types of scope:

- **global scope** (sometimes termed file scope) in which variables are declared outside of all functions and statement blocks, and
- **block scope** in which variables are declared within a function or statement block.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

static int count = 0;

int main(int argc, char *argv[])
{
    int nfound = 0;
    // check the number of arguments
    if(argc != 2) {
        int nerrors = 1;
        ....
        exit(EXIT_FAILURE);
    } else {
        int ntimes = 100;
        ....
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

- The variable `count` has **global scope**.
  
  It is defined on line 06, and may be used anywhere from line 06 until the end of the file (line 26).
  
  The variable `count` is also preceded by the keyword `static`, which prevents it from being 'seen' (read or written) from outside of this file `rot.c`.

- The variable `nfound` has **block scope**.
  
  It is defined on line 10, and may be used anywhere from line 10 until the end of the block in which it was defined (until line 26).

- The variable `nerrors` has **block scope**.
  
  It is defined on line 14, and may be used anywhere from line 14 until line 18.

- The variable `ntimes` has **block scope**.
  
  It is defined on line 20, and may be used anywhere from line 20 until line 24.

We could define a different variable named `nerrors` in the block of lines 20-24 - without problems.

We could define a different variable named `nfound` in the block of lines 20-24 - but this would be a very bad practice!