The structure of C programs

Let's look at the high-level structure of a short C program, *rotate.c* (we're using ellipsis to omit some statements, for now). At this stage it's not important what the program is supposed to do.

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

/* Compile this program with:
   cc -std=c11 -Wall -Werror -o rotate rotate.c
 */

#define ROT 13

static char rotate(char c)
{
    c = c + ROT;
    .....
    return c;
}

int main(int argcount, char *argvalue[])
{
    // check the number of arguments
    if(argcount != 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.

  C11 provides two types of comments -

  1. /* block comments */ and
  2. // 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=c11 -Wall -Werror -o rotate rotate.c
 */

#define ROT 13

static char rotate(char c)
{
    c = c + ROT;
    .....
    return c;
}

int main(int argcount, char *argvalue[])
{
    // check the number of arguments
    if(argcount != 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.
- Most C programs you read will name *main()'s* parameters as *argc* and *argv*.
- 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.

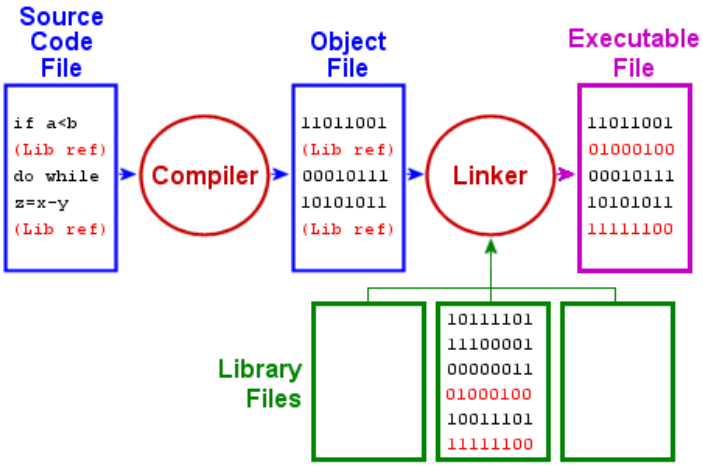
Compiling and linking our C programs

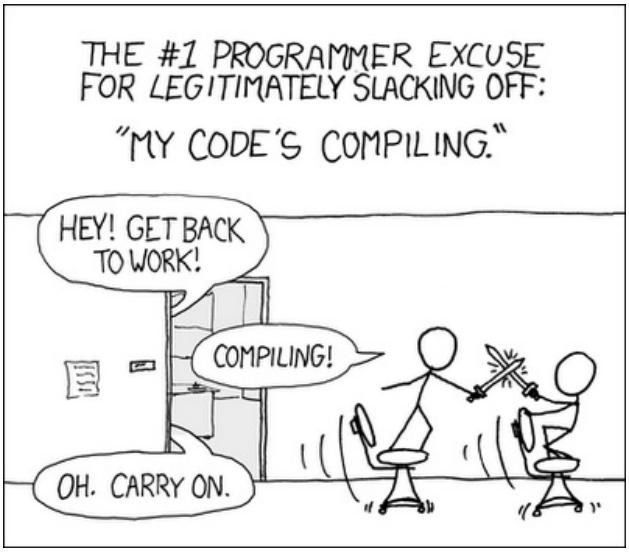
C programs are *human-readable text files*, that we term *source-code files*.

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 8-32GB of memory, or eight to thirty-two *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.

- 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).
- C variable names are *case sensitive*, thus:

  `MYLIMIT`, `mylimit`, `Mylimit` and `MyLimit`

  are four different variable names.
- 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`, `BUFSIZ`, `and` `ROT`
- Older C compilers limited variable names to, say, 8 unique characters. Thus, for them,

  `turn_nuclear_reactor_coolant_on` and `turn_nuclear_reactor_coolant_off`

  are the same variable! Keep this in mind if ever developing *portable* code for old environments.

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:

typename	description, and an example of variable initialization
bool	Boolean (truth values), which may only hold the values of either **true** or **false** *e.g.* **bool** finished = **false**;
char	character values, to each hold a single values such as an alphabetic character, a digit character, a space, a tab... *e.g.* **char** initial = 'C';
int	integer values, negative, positive, and zero *e.g.* **int** year = 2006;
float	floating point values, with a typical precision of 10 decimal digits (on our lab machines) *e.g.* **float** inflation = 5.1;
double	"bigger" floating point values, with a typical precision of 17 decimal digits (on our lab machines) *e.g.* **double** pi = 3.1415926535897932;

Some textbooks will (too quickly) focus on the actual storage size of these basic types, and emphasise 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 Significance of Integers in C

Throughout the 1950s, 60s, and 70s, there were many more computer hardware manufacturers than there are today. Each company needed to promote its own products by distinguishing them from their competitors.

At a low level, different manufacturers employed different memory sizes for a basic character - some just 6 bits, some 8 bits, 9, and 10. The unfortunate outcome was the incompatability of computer programs and data storage formats.

The C programming language, developed in the early 1970s, addressed this issue by *not* defining the required size of its datatypes. Thus, C programs are portable at the level of their source code - porting a program to a different computer architecture is possible, provided that the programs are compiled on (or for) each architecture. The only requirement was that:

sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)

Since the 1980s, fortunately, the industry has agreed on 8-bit characters or *bytes*. But (compiling and) running the C program on different architectures:

```c
#include <stdio.h>

int main(void)
{
    printf("char  %lu\n", sizeof(char));
    printf("short %lu\n", sizeof(short));
    printf("int   %lu\n", sizeof(int));
    printf("long  %lu\n", sizeof(long));
    return 0;
}
```

may produce different (though still correct) results:

```
char  1
short 2
int   4
long  8
```

It's permissible for different C compilers on different architectures to employ different sized integers.

Why does this matter? Different sized integers can store different maximum values - the above datatypes are *signed* (supporting positive and negative values) so a 4-byte integer can only represent the values -2,147,483,648 to 2,147,483,647.

If employing integers for 'simple' counting, or looping over a known range of values, there's rarely a problem. But if using integers to count many (small) values, such as milli- or micro-seconds, it matters:

- Fun fact: GPS uses 10 bits to store the week. That means it runs out... oh heck ? April 6, 2019
- Airlines Have To Reboot Their Airbus A350 Planes After Every 149 Hours
- To keep a Boeing Dreamliner flying, reboot once every 248 days

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.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
04  #include <ctype.h>
05
06  static int count = 0;
07
08  int main(int argcount, char *argvalue[])
09  {
10      int nfound = 0;
11
12      // check the number of arguments
13      if(argcount != 2) {
14          int nerrors = 1;
15
16          ....
17          exit(EXIT_FAILURE);
18      }
19      else {
20          int ntimes = 100;
21
22          ....
23          exit(EXIT_SUCCESS);
24      }
25      return 0;
26  }
```

- 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 *rotate.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!

Flow of control in a C program

A program's *control flow* describes how sequences of statements are executed.
Flow control in a C program is very similar to most other imperative and object-oriented languages.

- C programs commence their execution at their *main()* function, execute their statements, and exit (return the flow of control) to the operating system.
- It's fairly obvious that statements *need* to be executed in a well-defined order, as we expect programs to always behave the same way (unless some *random data* directs the execution path, as in computer games, simulations, and heuristic algorithms).
- Default flow of control executes each statement in order, top-to-bottom.
- Programs that only execute from top-to-bottom are pretty boring, and we need to control their flow with a variety of *conditional* statements, *loops*, and *function-calls*.

Conditional execution

Conditional statements first *evaluate* a Boolean condition and then, based on whether it's true or false, execute other statements.

The most common form is:

```
if(condition1) {
  // more statements;
  .....
}
else {
  // more statements;
  .....
}
```

Sometimes, the **else** *clause* is omitted:

```
if(condition1) {
  // more statements;
  .....
}
```

Often, the **else** *clause* provides further **if** statements:

```
if(condition1) {
  // more statements;
  .....
}
else if(condition2) {
  // more statements;
  .....
}
else {
  // more statements;
  .....
}
```

Note that in the examples, above, each *block of statements* to be executed has been written within curly-brackets.

The curly-brackets are *not required* (we could just write a single statement for either **if** or **else** clause). However, adding curly-brackets is *considered a good practice*. They provide a safeguard for when additional statements are added later.

Boolean values

Of significance, and a very common cause of errors in C programs, is that C standards, prior to ISO-C99, had no Boolean datatype.

Historically, an integer value of zero evaluated equivalent to a Boolean value of *false*; any non-zero integer value evaluated as *true*.

You may read some older C code:

```
int initialised = 0; // set to false
....

if(! initialised) {
  // initialisation statements;
  .....
  initialised = 1; // set to true
}
```

which may be badly and accidently coded as:

```
int initialised = 0; // set to false
....

if(initialised = 0) {
  // initialisation statements;
  .....
  initialised = 1; // set to true
}
```

so, employ defensive programming:

```
int initialised = 0; // set to false
....

if(0 = initialised) { // invalid syntax!
  // initialisation statements;
  .....
  initialised = 1; // set to true
}
```

In the second example, the conditional test *always* evaluates to *false*, as the single equals character requests an assignment, not a comparison.

It *is* possible (and occassionally reasonable) to perform an assignment as part of a Boolean condition - you'll often see:

```
while( (nextch = getc(file) ) != EOF ) {....
```

Whenever requiring the **true** and **false** constants (introduced in C99), we need to provide the line:

```
#include <stdbool.h>
```

Switch statements

When the same (integer) expression is compared against a number of distinct values, it's preferred to evaluate the expression once, and compare it with possible values:

Cascading **if..else..if..** statements:

```
if(expression == value1) {
  // more statements;
  .....
}
else if(expression == value2) {
  // more statements;
  .....
}
else {
  // more statements;
  .....
}
```

The equivalent **switch** statement:

```
switch (expression) {
  case value1 :
  // more statements;
    .....
    break;

  case value2 :
    // more statements;
    .....
    break;

  default :
    // more statements;
    .....
    break;
}
```

Less-common features of the **switch** statement:

```
switch (expression) {
  case value1 :
  case value2 :
    // handle either value1 or value2
    .....
    break;

  case value3 :
    // more statements;
    .....
    // no 'break' statement, drop through

  default :
    // more statements;
    .....
    break;

}
```

- Typically the 'expression' is simply an identifier, but it may be arbitrarily complex - such as an arithmetic expression, or a function call.
- The datatype of the 'expression' must be an integer (which includes characters, Booleans, and enumerated types), but it cannot be a real or floating-point datatype.
- The **break** statement at the end of each **case** indicates that we have finished with the 'current' value, and control-flow leaves the **switch** statement.
  Without a **break** statement, control-flow continues "downwards", flowing into the next **case** branch (even though the expression does not have that **case**'s value!).
- **switch** statements with 'dense' values (none, or few integer missing) provide good opportunities for optimised code.
- There is no need to introduce a new block, with curly-brackets, *unless* you need to define new local variables for specific case branches.

Flow of control in a C program - bounded loops

One of the most powerful features of computers, in general, is to perform thousands, millions, of repetitive tasks quickly

(in fact, one of the motivating first uses of computers in the 1940s was to calculate trigonometric tables for the firing of artillery shells).

C provides its **for** control statement to *loop* through a sequence of statements, a *block* of statements, a known number of times:

The most common form appears below, in which we introduce a *loop control variable*, *i*, to count how many times we go through the loop:

```
// here, variable i holds the values 1,2,...10

for(int i = 1 ; i <= 10 ; i = i+1) {
// the above introduced a loop-control variable, i
  .....
  printf("loop number %i\n", i);
  .....
// variable i is available down to here
}

// but variable i is not available from here
```

The loop control variable does not always have to be an integer:

```
// here, variable ch holds each lowercase value

for(char ch = 'a' ; ch <= 'z' ; ch = ch+1) {
  .....
  printf("loop using character '%c'\n", ch);
  .....
}
```

Notice that in both cases, above, we have introduced *new* variables, here *i* and *ch*, to specifically *control* the loop.

The variables may be used *inside* each loop, in the statement block, but then "disappear" once the block is finished (after its bottom curly bracket).

It's also possible to use any other variable as the loop control variable, even if defined *outside* of the **for** loop. In general, we'll try to avoid this practice - *unless* the value of the variable is *required* outside of the loop.

Flow of control in a C program - unbounded loops

The **for** loops that we've just seen should be used when we know, ahead of time, how many times we need to loop (i.e. 10 times, or over the range 'a'..'z').

Such loops are termed *bounded* loops and, unless we've made an unseen coding error, always terminate after a fixed number of iterations.

There are also many occasions when we *don't* know, ahead of time, how many iterations may be required. Such occasions require *unbounded* loops.

C provides two types of unbounded loop:

The most common is the **while** loop, where zero or more iterations are made through the loop:

```
#define  NLOOPS    20

int i = 1;
int n = 0;
 .....

while(i <= NLOOPS) {
    printf("iteration number %i\n", i);
    .....
    .....
    i = some_calculation_setting_i;
    n = n + 1;
}

printf("loop was traversed %i times\n", n);
```

Less common is the **do....while** loop, where at least one iteration is made through the loop:

```
#define  NLOOPS    20

int i = 1;
int n = 0;
 .....

do {
    printf("iteration number %i\n", i);
    .....
    .....
    i = some_calculation_setting_i;
    n = n + 1;
} while(i <= NLOOPS);

printf("loop was traversed %i times\n", n);
```

Notice that in both cases we still use a variable, *i*, to control the number of iterations of each loop, and that the changing value of the variable is used to determine if the loop should "keep going".

However, the statements used to modify the control variable may appear almost anywhere in the loops. They provide flexibility, but can also be confusing when loops become severals tens or hundreds of lines long.

Notice also that **while**, and **do....while** loops cannot introduce new variables to control their iterations, and so we have to use existing variables from an outer lexical scope.

Writing loops within loops

There's a number of occassions when we wish to loop a number of times (and so we use a **for** loop) and *within* that loop we wish to perform another loop. While a little confusing, this construct is often quite common. It is termed a *nested loop*.

```
#define  NROWS     6
#define  NCOLS     4

for(int row = 1 ; row <= NROWS ; row = row+1) {       // the 'outer' loop

    for(int col = 1 ; col <= NCOLS ; col = col+1) {   // the 'inner' loop
        printf("(%i,%i)  ", row, col);                // print row and col as if "coordinates"
    }
    printf("\n");                                     // finish printing on this line
}
```

The resulting output will be:

```
(1,1)  (1,2)  (1,3)  (1,4)
(2,1)  (2,2)  (2,3)  (2,4)
(3,1)  (3,2)  (3,3)  (3,4)
(4,1)  (4,2)  (4,3)  (4,4)
(5,1)  (5,2)  (5,3)  (5,4)
(6,1)  (6,2)  (6,3)  (6,4)
```

Notice that we have two distinct loop-control variables, `row` and `col`.

Each time that the *inner loop* (`col`'s loop) starts, `col`'s value is initialized to 1, and advances to 4 (NCOLS).

As programs become more complex, we will see the need for, and write, all combinations of:

- **for** loops within **for** loops,
- **while** loops within **while** loops,
- **for** loops within **while** loops,
- and so on....

Changing the regular flow of control within loops

There are many occasions when the default flow of control in loops needs to be modified.

Sometimes we need to leave a loop early, using the **break** statement, possibly skipping some iterations and some statements:

```
for(int i = 1 ; i <= 10 ; i = i+1) {
    // Read an input character from the keyboard
    .....
    if(input_char == 'Q') { // Should we quit?
        break;
    }
    .....
    .....
}
// Come here after the 'break'.  i is unavailable
```

Sometimes we need to start the *next* iteration of a loop, even *before* executing all statements in the loop:

```
for(char ch = 'a' ; ch <= 'z' ; ch = ch+1) {
    if(ch == 'm') { // skip over the character 'm'
        continue;
    }
    .....
    .....
    statements that will never see ch == 'm'
    .....
    .....
}
```

In the first example, we iterate through the loop at most 10 times, each time reading a line of input from the keyboard. If the user indicates they wish to quit, we **break** out of the bounded loop.

In the second example, we wish to perform some work for all lowercase characters, *except* 'm'.
We use **continue** to ignore the following statements, and to start the next loop (with ch == 'n').

The equivalence of bounded and unbounded loops

We should now be able to see that the **for**, **while**, and **do ... while** control flow statements are each closely related.

To fully understand this, however, we need to accept (for now), that the three "pieces" of the **for** construct, are not always *initialization, condition, modification*.

More generally, the three pieces may be C *expressions* - for the moment we'll consider these as C *statements* which, if they produce a value, the value is often ignored.

The following loops are actually equivalent:

```
for( expression1 ; expression2 ; expression3 ) {
    statement1;
    ....
}
```

```
expression1;
while(expression2) {
    statement1;
    ....
    expression3;
}
```

In both cases, we're expecting *expression2* to produce a Boolean value, either **true** or **false**, as we need that truth value to determine if our loops should "keep going".

You should think about these carefully, perhaps perform some experiments, to determine where control flow really goes when we introduce **break** and **continue** statements.

Some unusual loops you will encounter

As you read more C programs written by others, you'll see some statements that *look* like **for** or **while** loops, but appear to have something missing.
In fact, any (or all!) of the 3 "parts" of a **for** loop may be omitted.

For example, the following loop initially sets `i` to 1, and increments it each iteration, but it doesn't have a "middle" conditional test to see if the loop has finished. The missing condition constantly evaluates to **true**:

```
for(int i = 1 ; /* condition is missing */ ; i = i+1) {
    .....
    .....
}
```

Some loops don't even have a loop-control variable, and don't test for their termination. This loop will run *forever*, until we *interrupt* or *terminate* the operating system process running the C program.
We term these *infinite loops* :

```
// cryptic - avoid this mechanism
for( ; ; ) {
    .....
    .....
}
```

```
#include <stdbool.h>

// clearer - use this mechanism
while( true ) {
    .....
    .....
}
```

While we often see and write such loops, we don't usually want them to run forever!

We will typically use an enclosed condition and a **break** statement to terminate the loop, either based on some user input, or the state of some calculation.

Lecture 2 Summary

- C11 is a *small* programming language (in terms of its number of keywords), although some aspects can be difficult to learn when compared to newer languages.
- C is a *compiled* programming language. The human-readable source code of a C program needs translating to executable machine code before the program can be executed.
- Unlike Python programs, white-space is insignificant and is not used to define lexical scoping. New lexical blocks are introduced with curly brackets.
- Alphabetic case is significant. Any colour seen in textbooks or possibly (visually) added by your text editor, is insignificant.
- C provides a number of integer data types, of differing system-dependent widths, and two floating-point data types. Data types of exact widths may be explictly specified for architecture-specific programming.
- Floating-point data types are only *very* rarely used in systems programming.
- C's control-flow statements - conditional statements, bounded and unbounded loops, and nested instances of these, and unconditional control-flow - execute similarly to equivalent statements in other languages, although their syntax may appear unusual.