Brief Summary of C

Statements in C end with a semi-colon (`;`). C treats all blanks space as equivalent, so line breaks and indents are for readability only. Blocks of code are surrounded with braces, `{` and `}`. A single statement without the braces also counts as a block of code.

The basic data types in C are `int` and `char`. An `int` holds an integer value, which can be 2, 4 or 8 bytes depending on the platform (none of the code here depends on the exact length of an `int`). A `char` holds a single byte. Single characters are surrounded by single quotes, such as `'a'` and `'x'`. There are other data types for floating point numbers and integers of different sizes, which I won't use.

Variable and function names are case-sensitive. Variables are declared with the type followed by the name, for example:

```
int counter;
```

and multiple variables can be declared together, separated by a comma:

```
char letter, lastbyte, direction;
```

Arrays in C are denoted with square brackets and indexed from 0, so

```
int scores[20];
```

allocates room for 20 `int`s, of which `scores[0]` would be the first and `scores[19]` the last.

Assignment is done with the = sign:

```
counter = 0;
```

Variables can be declared and initialized in one step:

```
int bytecount = 0;
```

Arithmetic is as expected, with expressions grouped using parenthesis:

```
counter = counter + 1;
lastbyte = ((direction - 5) * 6) / 2;
```

The statement

```
++counter;
```

is shorthand for

```
counter = counter + 1;
```

Strings are simply arrays of type char. By convention, a string is terminated with a 0 value, written as a single character `'\0'`. Thus the length of a string may be less than the size of the char array it is stored in. Declaring

```
char name[10];
```

allocates room for a string that can be up to 9 bytes long, since one byte must be left for the terminating `'\0'` (you could put a different character in the tenth byte, but it would then not be a properly-terminated string according to C conventions). The code

```
name[0] = 't';
name[1] = 'e';
name[2] = 'd';
name[3] = '\0';
```

will set the name to be `"ted"`, with the 6 extra bytes unused at that point. A string in double quotes, such as `"hello"`, is converted by the compiler into a char array included the final `'\0'`, so `"hello"` occupies six bytes.

Pointers are declared with `*`, for example:

```
char * city;
```

which only allocates storage for the pointer itself. Pointer can be declared together with variables of the type, so

```
char * city, name;
```

declares a pointer to a `char` called `city,` and a `char` (*not* a pointer) called `name`. `char` pointers are often assigned to constant strings, for example

```
city = "Boston";
```

which will automatically allocate the 7 bytes needed to store the string `"Boston"` and set `city` to point to it.

The value `NULL` is assigned to pointers to indicate that they point to nothing.

Pointers are also dereferenced with `*`, so `*city` is the first byte pointed to by `city`. In fact, pointers and arrays are often used interchangeably, and the first char in the `city` array could be referenced as `city[0]` or `*city`. Note that C does no checks for validity of pointers, so `*city` will likely cause a crash if `city` is uninitialized, and `name[20]` gives an undefined result if `name` is allocated as above with room for only 10 `chars`.

Pointer arithmetic is allowed and automatically compensates for the size of the element pointed to. Thus, `city+2` will point to two bytes after `city`, since a `char` occupies one byte, but for an `int` array declared as

```
int distances[5];
```

and assuming an `int` occupies 4 bytes, `distances+2` will be 8 bytes after distances. More generally, `array[n]` is equivalent to `*(array + n)` and in fact is defined as such.

Structures are defined as follows:

```
typedef struct _record {
    int element1;
    char element2;
    struct _record * next;
} record, * record_ptr;
```

This combines two things (which could be separated if desired, but won't be in this book): the definition of the structure `_record`, and the creation of a new type `record` which is equivalent to the more cumbersome `struct _record` (it also defines a new type `record_ptr`, a pointer to a `record`). Within the structure definition itself, `struct _record` is used because the `typedef` is not finished, but from then on `record` can be used instead.

Variables can then be declared such as:

```
record current_record;
record_ptr first_record;
```

For clarity, in this book programs will use `record *` as opposed to `record_ptr` to indicate a pointer to a `record`. `record_ptr *` means a pointer to a pointer to a `record`.

Elements in a struct are referenced with `.`, as in:

```
current_record.element1
```

For pointers `->` is used to combine dereferencing a pointer to a structure and accessing an entry in the structure, as in:

```
first_record->element2;
```

which is equivalent to:

```
(*first_record).element2;
```

or even:

```
first_record[0].element2;
```

Conditional statements are defined as:

```
if (test-expression)
    true-code-block
else
    false-code-block
```

with the `else` and `false-code-block` optional. The code blocks can a single statement, or multiple statements surrounded by braces. The `if()` is true if test-expression evaluates to a non-zero value, false if it is zero. Comparisons are done with ==, `!=`, <, >, <=, and >=.

In C, an assignment statement is also an expression having the value of the left-hand of the assignment, so the assignment statement

```
c = 5
```

evaluates to `5` and you could write:

```
d = (c = 5);
```

The ++ operator, seen earlier, can be written before or after the variable; when written before the value returned by the assignment is the new value, but when written after, the old value is returned:

```
j = 5;
k = ++j;   // k will be 6
m = k++;    // m will also be 6
```

There is also a `--` operator that works the same way for subtracting 1.

It is a common mistake in C to write

```
if (c = 5)
```

(which will always evaluate to `5`, thus always non-zero and always true) instead of

```
if (c == 5)
```

which evaluates as expected, true if `c` is equal to `5`, false otherwise. Finally,

```
if (c)
```

is the same as

```
if (c != 0)
```

Loops can be done with a `for` statement:

```
for (init-statement ; test-expression ; increment-statement )
    for-code-block
```

Typically `init-statement` initializes a loop counter, `test-expression` is an expression involving the loop counter, and `increment-statement` modifies the loop counter (but that is not always true):

```
int array[20];
for (i = 0; i < 20; i++) {
    { code involving array[i]; }
}
```

This walks through the elements of array. Note that `test-expression` is `i < 20`, not `i <= 20`, since entries in an array of size 20 are accessed as `i[0]` through `i[19]`.

`test-expression` is evaluated at the beginning of each iteration through the loop, and if it is true (non-zero), `for-code-block` is executed. At the end of the loop, `increment-statement` is executed. From anywhere within a loop, the statement `continue` will jump to the end of the loop (causing `increment-statement` to execute and then beginning another check of `test-expression` and possible iteration of the loop); the statement `break` will leave the loop immediately, without executing `increment-statement`.

There is also a `while` loop:

```
while (test-expression)
    while-code-block
```

which evaluates `text-expression` each time, and executes `while-code-block` if it is true. `continue;` and `break;` can also be used within `while` loops.

Functions are defined as:

```
return-type function-name(type1 argument1, type2 argument2)
{
    local-variable-declarations;
    function-code;
}
```

If `argument1` is an array, it is followed with `[]`, as in the example

```
int find_largest ( int array[], int array_length )
```

`local-variable-declarations` consists of variables declarations that are local to that function.

The `return` statement exits a function. `return` should be followed by a variable of the proper `return-type` for the function. A special `return-type` of `void` in the function declaration means the function does not return a value and the `return` statement needs no arguments. Functions that return type `void` can end without a `return` statement.

C code is run through a pre-processor before it is compiled. The main way in which programmers are aware of the pre-processor is that it can substitute constant definitions throughout the code; so for example the pre-processor statement

```
#define  ARRAY_SIZE  20
```

will cause the pre-processor to substitute `20` every place it sees `ARRAY_SIZE`. `#define` can be used to define functions with arguments that are replaced, such as

```
#define  NEGATIVE(x)  (-(x))
```

but we won't use that in the examples.

Comments are denoted by `//`; everything after that on a line is ignored (one of the rare cases in C where a line break has a different meaning from other white space, since only a line break will end a `//` comment). Comments can also be delimited by a starting `/*` and an ending `*/`; within those comments a line break is like any other white space, and has no effect on the comment.