Nature of C

C tends to trust the programmer, and has few rules. You have the liberty to write unusual code, but that makes it easier to write needlessly tricky code.

Essentials

Source files are compiled into object files, which are machine-independent. These are linked in using the linker, along with the runtime library, to produce executable code.

The library can be statically or dynamically linked.

You can use `#include` to include header files from other source code files, which contain function definitions, etc.

Function Definition

Functions need to be defined before they are called, or else you need to tell the compiler that the function exists somewhere, using the "extern" keyword:

```
int main() {
    extern int square(int);
}
```

You're telling the compiler that the function will be found by the linker.

`main()`

`main()` can't take arguments, but can take arguments from the command line.

Function Polymorphism

C doesn't have the function overloading we're familiar with from Java, but you can have functions that take variable numbers of parameters.

Every function must have a unique name.

`exit(0)`

The exit function (or return) should be called in every main function, and typically an exit value of 0 indicates a successful run. Non-zero values indicate different errors.

`scanf()`

This is the mirror of `printf()`, and reads from the standard input into variables.

To tell the function where to put the value, you need to use &, the address-of operator. This gives `scanf()` an address to put the input value into.

```
scanf("%d", &input_val);
```

The Preprocessor

This is run before the compiler, and allows e.g. some text manipulation.

`#include`

Two forms, angle brackets and quotes.

- angle brackets form tells the compiler to look in the library
- quotes form tells the compiler to look in a couple of places:
  - the folder this code was in
  - then the path
  - then the library

This takes a file and copies its text into the current file.

Scalar Data Types

You can find the address of an "object" (e.g. a variable) by using the & operator:

```
long long_var = 3;
long *ptr = &long_var;
```

The asterisk makes this variable a pointer, which holds a memory address (in this case of a long). The & operator gets the address of `long_var` to put it in the `ptr` variable.

Dereferencing a pointer is done with an asterisk:

```
/* p_ch is a pointer to a character - the memory address of a character. */
char* p_ch;
char ch1 = 'A', ch2;

/* p_ch is now used to store the memory address of ch_1. */
p_ch = &ch1;
/* ch_2 is now assigned the value stored at the address p_ch holds. */
ch_2 = *p_ch
```

The dereferenced pointer can also appear on the left-hand side of an assignment:

```
*fp = 3.15;
```

This will update the value at the address fp points to.

You can have a pointer to most things (e.g. a function which you can dereference and call).

Types

You can make your own types in C by using the `typedef` keyword, making synonyms for types:

```
typedef long int FOUR_BYTE_INT;
```

Implicit Conversions

In assignment statements, the value on the right-hand side is converted to the type of the variable on the left-hand side.

When a `char` or `short int` appears in an expression, it's converted to an `int`. `unsigned chars` and `unsigned shorts` are converted to `int` if it's big enough to hold their value – otherwise they're converted to[...]

[...]

Arrays and Pointers

In C, arrays are just space reserved for a number of values, and a special pointer.

```
/* Create an array that can hold 365 integers: */
int daily_temp[365];
```

You can get the size of the array with the `sizeof` command, which gets the size of the array in bytes. You can divide this by the size of one of the elements of the array to give the number of elements in the array:

```
sizeof(array) / sizeof(array[0]);
```

You can initialise arrays:

```
static int[5] array_a = {1, 2, 3, 4, 5};
static int[5] array_b = {1, 2, 3};
static int[] array_c = {1, 4, 5, 5};
```

Pointer Arithmetic

In C you can add/subtract integers to/from pointers:

```
long *p1, *p2;
```

p1 + 3 finds an address that's 3 longs beyond p1. If a long is 8 bytes, this will add 24 to p1. This is known as pointer scaling.

Subtracting two pointers that point to the same type of object gives the number of objects between the two pointers:

```
&a[3] - &a[0]; /* evaluates to 3 */
```

Null Pointer

The null pointer is guaranteed not to point to a valid object. It's assigned the integral value 0, and is useful in control-flow statements, as it evaluates to false.

Passing Pointers as Function Arguments

The compiler will complain if you try to mix different types of pointer.

However, in the absence of function prototyping, the compiler doesn't check if the type of the actual argument is the same as the type of the formal argument.[?]

Accessing Array elements through Pointers

```
p = &array[0];
```

p is now a pointer to the first element in the array, and `*p` is equivalent to `array[0]`.

You can add to this pointer to access other elements in the array:

```
*(p + 3); /* equivalent to array[3] */
```

Note: `array` is a static pointer created by the compiler (meaning you can't change it). Since it's a pointer, though, you can rewrite the earlier block as:

```
p = array;

*(array + 3);
```

Strings

A string is an array of characters terminated by a null character (a character with a numeric value of zero – `\0`). Each character in the string takes up one byte, and the compiler automatically adds the null character, so the array is one element longer than the number of characters in the string.

If you specify the size of the array, you need to allocate enough chars to hold the string.

Note: using double quotes for string literals is just syntactic sugar – you can also use an array literal containing characters.

String literals are copied into RAM when the program starts running. You can run out of RAM on a microcontroller if you have a lot of them in your code. There's a macro which allows you to store strings in the flash memory, but then you need to deal with getting them from memory.

You can also initialise a char pointer with a string constant:

```
char *ptr = "more text";
```

The pointer declaration creates an additional 4-byte variable for the pointer. The pointer is a variable that is initialised with the address of the array's initial element. This pointer is not a constant pointer (like you would get with an array), so if you reassign it you'll lose it.

Multidimensional Arrays

Use arrays in arrays:

```
int x[3][4][5];
```

Here, `x` is a 3-element array. Each element in `x` is a 4-element array, and every element of those 4-element arrays is a 5-element array.

Pointers to Pointers

Using more `*`s, you can have pointers to pointers:

```c
int r = 5;
int *q = &r;
int **p = &q;

/* These 3 statements are equivalent: */
r = 10;
*q = 10;
**p = 10;
```