The char Type

The C `char` type stores small integers. It is usually 8 bits.
It is guaranteed to be able to store integer values in the interval 0 to +127.
It is mostly used to store characters encoded in ASCII.
Characters are the only thing you should store in `char` variables in COMP1911.
For example even if a numeric variable only containing the values 0..9 use the type `int` for the variable.

ASCII Encoding

ASCII (American Standard Code for Information Interchange) specifies encoding for 128 characters to the integers 0..127.
The characters encoded include upper and lower case (English) letters, digits, common punctuation symbols (as you might find on the keyboard) plus a series of non-printable control characters (including newline and tab).

Manipulating Characters

Does this mean you will have to remember 100+ character codes!?
Luckily no! We use *character literals* instead!

For example:

```
char a = 'a';          // ASCII code 97
char A = 'A';          // ASCII code 65
char space = ' ';      // ASCII code 32
char plus = '+';       // ASCII code 43
char newline = '\n'; // ASCII code 10
```

Style Note

Always use character literals in your code! Even if you are really proud of having memorised the ASCII Table!

Manipulating Characters

NOTE

The ASCII codes for the digits (48–57), the upper case letters (65–90) and lower case letters (97–122) are in sequence.

Knowing this allows us to do some neat things:

```
int a = 'a';
int b = a + 1;    // this is possible due to
int c = 'a' + 2; // the underlying numeric type
int B = b - ('a' - 'A');
```

Manipulating Characters

We can also test various properties of characters:

```
// check for lowercase
if (c >= 'a' && c <= 'z') {
   ...
```

Problem: Convert a digit character to the integer it represents, e.g., '0' $\mapsto$ 0, '7' $\mapsto$ 7, etc.

We use the fact that the digits codes are in order:

```
// check is a digit
if (c >= '0' && c <= '9') {
   val = c - '0'; // why does this work?
}
```


Printing and Reading Characters

C provides library functions for reading and writing characters

The `getchar` function

This function reads and returns one input character. Note that, unlike `scanf`, it has no arguments; its return value is collected by assigning it to a variable.

The `putchar` function

This function takes a single `int` argument and prints it out

Here is an example:

```
int c;
printf("Please enter a character: ");
c = getchar();
putchar(c);
```


Reading Characters

Consider the following code:

```
int c1,c 2;

printf("Please enter first character:\n");
c1 = getchar();
printf("Please enter second character:\n");
c2 = getchar();
printf("First %c\nSecond: %c\n", c1, c2);
```

What is the output? Turns out that the newline input by pressing *Enter* after the first character is read by the second `getchar`.

Reading Characters

How can we fix the program?

```
int c1, c2;

printf("Please enter first character:\n");
c1 = getchar();
getchar(); // reads and discards a character
printf("Please enter second character:\n");
c2 = getchar();
printf("First: %c\nSecond: %c\n", c1, c2);
```

Printing characters

The conversion specifier for characters is `%c`. Using it we can supply variables of `char` type to `printf` for output:

End of Input

An input funcion can such as `scanf` or `getchar` can fail because there is no input is available.
This can occur, for example, if input is coming from a file and the end of the file is reached.
On UNIX-like systems such Linux & OSX typing **Ctrl + D** on a terminal signals to the operating system there is no more input from the terminal. Windows has no equivalent, but some windows program interprert **Ctrl + Z** similarly.
`getchar` returns a special non-ASCII value to indicate there is no input was available.
This non-ASCII value is #defined as `EOF` in stdio.h.
On most systems EOF == -1. Note -1, isn't an ASCII value (0..127)
There is no end-of-file character on Linux or other modern operating systems.

Reading Characters to End of Input

Programming pattern for reading characters to the end of input:

```
int ch;

ch = getchar();
while (ch != EOF) {
    printf("you entered the character: '%c' which ha
    ch = getchar();
}
```

For comparison the programming pattern for reading integers to end of input:

```
int num;
// scanf returns the number of items read
while (scanf("%d", &num) == 1) {
    printf("you entered the number: %d\n", num);
}
```


Strings

A string is a sequence of characters.
C uses a special ASCII character '\0' to mark the end of strings.
This is convenient because programs don't have to track the lnegth of the string.

Definition

A C string is a null-terminated character array.

Consider the following:

```
// this is incorrect, '\0' will be discarded
char hello[5] = {'h', 'e', 'l', 'l', 'o', '\0'};

// this is OK
char hello[6] = {'h', 'e', 'l', 'l', 'o', '\0'};

// this is better
char hello[]  = {'h', 'e', 'l', 'l', 'o', '\0'};
```

String Length

The null character must be accounted for when sizing strings, although, somewhat confusingly, the library function `strlen` returns the length of a string not including the null character.

Useful C Library Functions for Characters

The C library includes some useful functions which operate on characters.
Several of the more useful listed below.
Note the you need to `#include <ctype.h>` to use them.

```
#include <ctype.h>

int toupper(int c); // convert c to upper case
int tolower(int c); // convert c to lower case

int isalpha(int c); // test if c is a letter
int isdigit(int c); // test if c is a digit
int islower(int c); // test if c is a lower case letter
int isupper(int c); // test if c is a upper case letter
```

Strings

Because working with strings is so common, C provides some convenient syntax.
Instead of writing:

```
char hello[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

You can write

```
char hello[] = "hello";
```

Note `hello` will have 6 elements The compiler automatically appends `'\0'` when strings are initialised with string literals.
Again, remember to allow space for it if sizing the array manually.

Reading an Entire Input Line

The function `fgets` reads an entire line:

```
#define MAX_LINE_LENGTH 1024
...
char line[MAX_LINE_LENGTH];

fgets(line, MAX_LINE_LENGTH, stdin);
fputs(line, stdout);
```


Reading an Entire Input Line

You might use `fgets` as follows:

```
#define MAX_LINE_LENGTH 1024
...
char line[MAX_LINE_LENGTH];

fgets(line, MAX_LINE_LENGTH, stdin);
fputs(line, stdout);  // equivalent to printf("%s",line)
```


Reading Lines to End of Input

Programming pattern for reading lines to end of input:

```
// fgets returns NULL if it can't read any characters
while (fgets(line, MAX_LINE, stdin) != NULL) {
    printf("you entered the line: %s", line);
}
```

String Manipulation

The header file `string.h` provides some useful string functions:

```
// string length (not including '\0')
size_t strlen(const char *s);
// string copy
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
// string concatenation/append
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
// string compare
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
// character search
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```


String Manipulation

```
#include <string.h>
...
char str1[100] = "Hello World!";
char str2[100];

strncpy(str2, str1, 100);       // copy str1 to str2
if (strcmp(str1, str2) == 0) { // case-sensitive compare
  printf("Strings match!\n");
}                               // append str1 to str2
strncat(str2, str1, 100 - (strlen(str2) + 1));
if (strcasecmp(str1, str2)) {  // case-insensitive compare
  printf("Strings do not match!\n");
}
printf("%d\n", strlen(str2));  // string length
```

Note that `strlen` does not count the null character!

String Manipulation

Remember

You can find out about what else is available in `string.h` by running `man string`.

When working with strings we use the null character as a guard:

```
char str1[100] = "Hello World!";
char str2[100];
int i;

// the following code copies str1 into str2
i = 0;                          // start at index 0
while (str1[i] != '\0') {      // stop on null
  str2[i] = str1[i];            // copy individual characters
  i = i + 1;                    // increment index
}
str2[i] = '\0';                 // MUST set this for str2!
```


Strings

In summary strings:

- are null-terminated character arrays
- can be initialised with string literals
- can be manipulated by `scanf`/`printf`, use `%s`
- benefit from the string manipulation functions in `string.h`
- since they are arrays they cannot be copied via assignment (`=`)

Careful

The main error encountered when working with strings is mishandling the terminating null character, e.g., forgetting to set it! Check this first if your strings are behaving strangely.

Arrays of Strings

Sometimes, instead of manipulating each individual cell, as for matrices, we need to manipulate whole arrays. This is generally the case when working with arrays of strings!

Consider:

```
char names[3][20] = {"Mark", "Luke", "John"};

// why does this work?
printf("%s %s %s\n", names[0], names[1], names[2]);
```

Array of arrays

If we take this view (array of arrays!) of 2D arrays, it makes sense why using only the first index gives us a whole array!

Command-line Arguments

What are command-line arguments? Arguments that are supplied to a program when it is run.

We have seen them already, for example:

```
% diff -i file1.txt file2.txt
% gedit prog.c
```

Here, `-i`, `file1.txt` and `file2.txt` are command-line arguments to `diff` and `prog.c` is a command-line argument to `gedit`.

Command-line arguments are automatically supplied to our C programs, by the operating system, via the arguments of the `main` function (`argc` and `argv`)!

```
int main(int argc, char *argv[]) { ...
```


Command-line Arguments

Arguments to `main`

`argc` stores the number of command-line arguments

`argv` stores the command-line arguments as strings

```
int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < argc; i = i + 1) { // print arguments
    printf("Argument %d is: %s\n", i + 1, argv[i]);
  }
  ...
```

NB

The first argument is always the program name! This means that `argc` is always at least 1 and `argv` contains at least one value.

Command-line Arguments

By default, command-line arguments are space delimited. We can use quotes if arguments include spaces.

```
% ./prog1 nospace one space
% ./prog2 nospace "one space"
```

In the above, `prog1` sees three command-line arguments, while `prog2` sees only two. What about?

```
% ./prog3 *.c
% ./prog4 10 < in > out
```

Sometimes `argv` is typed as: `char **argv`.