The char Type

- The C type `char` stores small integers.
- It is 8 bits (almost always).
- `char` guaranteed able to represent integers 0 .. +127.
- `char` mostly used to store ASCII character codes.
- Don’t use `char` for individual variables, only arrays.
- Only use `char` for characters.
- Even if a numeric variable is only use for the values 0..9, use the type `int` for the variable.
• ASCII (American Standard Code for Information Interchange)
• Specifies mapping of 128 characters to integers 0..127.
• The characters encoded include:
  ▶ upper and lower case English letters: A-Z and a-z
  ▶ digits: 0-9
  ▶ common punctuation symbols
  ▶ special non-printing characters: e.g. newline and space.

• You don’t have to memorize ASCII codes
  Single quotes give you the ASCII code for a character:

  ```c
  printf("%d", 'a');  // prints 97
  printf("%d", 'A');  // prints 65
  printf("%d", '0');  // prints 48
  printf("%d", ' ' + '\n');  // prints 42 (32 + 10)
  ```

• Don’t put ASCII codes in your program - use single quotes instead.
The ASCII codes for the digits, the upper case letters and lower case letters are contiguous. This allows some simple programming patterns:

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

// check is a digit
if (c >= '0' && c <= '9') {
    // convert ASCII code to corresponding integer
    numeric_value = c - '0';
}
```
Reading a Character - `getchar`

C provides library functions for reading and writing characters

- `getchar` reads a byte from standard input.
- `getchar` returns an int
- `getchar` returns a special value (EOF usually -1) if it can not read a byte.
- Otherwise `getchar` returns an integer (0..255) inclusive.
- If standard input is a terminal or text file this likely be an ASCII code.
- Beware input often buffered until entire line can be read.

```c
int c;
printf("Please enter a character: ");
c = getchar();
printf("The ASCII code of the character is %d\n", c);
```
Reading a Character - getchar

Consider the following code:

```c
int c1, c2;

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

The newline character from pressing Enter will be the second character read.
Reading a Character - getchar

How can we fix the program?

```c
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);
```
• Input functions such as `scanf` or `getchar` can fail because no input is available, e.g., if input is coming from a file and the end of the file is reached.

• On UNIX-like systems (Linux/OSX) typing `Ctrl + D` signals to the operating system no more input from the terminal.

• Windows has no equivalent - some Windows programs interpret `Ctrl + Z` similarly.

• `getchar` returns a special 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 `getchar` otherwise returns (0.255) or (0..127) if input is ASCII

• There is no end-of-file character on modern operating systems.
Reading Characters to End of Input

Programming pattern for reading characters to the end of input:

```c
int ch;
ch = getchar();
while (ch != EOF) {
    printf("\'%c\' read, ASCII code is %d\n", ch, ch);
    ch = getchar();
}
```

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

```c
int num;
// scanf returns the number of items read
while (scanf("%d", &num) == 1) {
    printf("you entered the number: %d\n", num);
}
```
• A string in computer science is a sequence of characters.
• In C strings are an array of `char` containing ASCII codes.
• These array of char have an extra element containing a 0.
• The extra 0 can also be written `'\0'` and may be called a null character or null-terminator.
• This is convenient because programs don’t have to track the length of the string.
Useful C Library Functions for Characters

The C library includes some useful functions which operate on characters. Several of the more useful listed below.

```c
#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 lower case letter
int isupper(int c); // test if c is upper case letter
```
Strings

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

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

You can write

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

Note `hello` will have 6 elements.
fgets - Read a Line

- **fgets(array, array_size, stream)** reads a line of text
  1. **array** - char array in which to store the line
  2. **array_size** - the size of the array
  3. **stream** - where to read the line from, e.g. stdin

- **fgets** will not store more than **array_size** characters in array
- Never use similar C function **gets** which can overflow the array and major source of security exploits
- **fgets** always stores a ‘\0’ terminating character in the array.
- **fgets** stores a ‘\n’ in the array if it reads entire line often need to overwrite this newline character:

```c
int i = strlen(lin);
if (i > 0 && line[i - 1] == "\n) {
    line[i - 1] = '\0';
}
```
You might use `fgets` as follows:

```c
#define MAX_LINE_LENGTH 1024
...
char line[MAX_LINE_LENGTH];
printf("Enter a line: ");

// fgets returns NULL if it can't read any character
if (fgets(line, MAX_LINE_LENGTH, stdin) != NULL) {
    fputs(line, stdout);
    // or
    printf("%s", line); // same as fputs
}
```
Programming pattern for reading lines to end of input:

```c
// fgets returns NULL if it can’t read any characters
while (fgets(line, MAX_LINE, stdin) != NULL) {
    printf("you entered the line: %s", line);
}
```
#include <string.h>

// string length (not including '\0')
int strlen(char *s);

// string copy
char *strcpy(char *dest, char *src);
char *strncpy(char *dest, char *src, int n);

// string concatenation/append
char *strcat(char *dest, char *src);
char *strncat(char *dest, char *src, int n);
#include <string.h>

// string compare
int strcmp(char *s1, char *s2);
int strncmp(char *s1, char *s2, int n);
int strcasecmp(char *s1, char *s2);
int strncasecmp(char *s1, char *s2, int n);

// character search
char *strchr(char *s, int c);
char *strrchr(char *s, int c);
Command-line arguments are 0 more strings specified when program is run.
If you run this command in a terminal:

```bash
$ dcc count.c -o count
```

dcc will be given 3 command-line arguments: "count.c" "-o" "count"

bf main needs different prototype if you want to access command-line arguments

```c
int main(int argc, char *argv[]) {
...
```
Accessing Command-line Arguments

- `argc` stores the number of command-line arguments + 1
- `argc == 1` if no command-line arguments
- `argv` stores program name + command-line arguments
  - `argv[0]` always contains the program name

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 1;
    printf("My name is \%s\n", argv[i]);
    while (i < argc) {
        printf("Argument \%d is: \%s\n", i, argv[i]);
        i = i + 1;
    }
}
```
Converting Command-line Arguments

`stdlib.h` defines useful functions to convert strings.

- `atoi` converts string to int
- `atof` converts string to double

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

int main(int argc, char *argv[]) {
    int i, sum = 0;
    i = 1;
    while (i < argc) {
        sum = sum + atoi(argv[i]);
        i = i + 1;
    }
    printf("sum of command-line arguments=%d\n", sum);
}
```