COMP1511: Characters, Strings, Finite Precision

Session 2, 2018

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 Encoding

- 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:

```
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.

ASCII Encoding Table

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Unicode

- ASCII is one of the most **common** format for text files so far.
- **Limitations**: ASCII, and its extensions, are **not** able to encode many world languages, they only supports english and few related european languages.

From wikipedia:

- **Unicode** is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.
- As of June 2018, Unicode 11.0 contains a repertoire of **137,439 characters** covering **146 modern and historic scripts**, as well as multiple ***symbol*** *sets* and ***emoji***.
- Many encoding schemes: UTF-8, UTF-16, and UTF-32. The most commonly used encodings are UTF-8, UTF-16.
- Takes more space compared to ASCII encoding. For example, UTF-32 (also referred to as UCS-4) uses **four bytes** for each character.

Manipulating Characters

The ASCII codes for the digits, the upper case letters and lower case letters are contiguous.

This allows some simple programming patterns:

```
// 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 bufferred until entire line can be read.

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

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 bufferred until entire line can be read.

```
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:

```
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);
```

End of Input

- 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 interprert **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:

```
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:

```
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 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.

```
#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 lette
int isupper(int c); // test if c is upper case lette
```

String

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.

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 ovewrite this newline character:

```
int i = strlen(lin);
if (i > 0 && line[i - 1] == "\n) {
    line[i - 1] = '\0';
}
```

Reading an Entire Input Line

You might use fgets as follows:

```
#define MAX_LINE_LENGTH 1024
...
char line[MAX_LINE_LENGTH];
printf"Enter a line: ");
// fgets returns NULL if it can't read any characte
if (fgets(line, MAX_LINE_LENGTH, stdin) != NULL {
        fputs(line, stdout);
        // or
        printf("%s" ,line); // same as fputs
}
```

Reading Lines to End of Input

Programming pattern for reading lines to end of input:

```
// fgets returns NULL if it can't read any character

while (fgets(line, MAX_LINE, stdin) != NULL) {
    printf("you entered the line: %s", line);
}
```

string.h

```
#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);
```

string.h

```c
#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

Command-line arguments are 0 more strings specified when program is run.
If you run this command in a terminal:

```
$ 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 comand-line arguments

```
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
`argv[1] argv[2] ...`	command-line arguments if supplied

```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
}
```

Finite Precision: Integer Representation

- typically 4 bytes used to store an **int** variable
- 4 bytes $\rightarrow$ 32 bits $\rightarrow$ 2^{32} possible values (bit patterns)
- only 2^{32} integers can be represented - which ones?
- -2^{31} to $2^{31}-1$
  i.e. -2,147,483,648 to +2,147,483,647
- Why are limits assymetric?
- zero needs a pattern (all zeros)
- can print bit values see:
  `https://cgi.cse.unsw.edu.au/~cs1511/code/C_basics/print_bits_of_int.c`
- More later and in COMP1521

Finite Precision: Integer Overflow/Underflow

- storing a value in an int outside the range that can be represented is illegal
- unexpected behaviour from most C implementations
  e.g the sum of 2 large positive integers is negative
- may cause programs to halt, or not to terminate
- can creates security holes
- bits used for **int** can be different on other platforms
- C on tiny embedded CPU in washing machine may use 16 bits
  -2^{15} to $2^{15} - 1$ i.e. -32,768 to +32767
- we'll show later how to handle this, for now assume 32 bit **ints**
- also arbitrary precision libraries available for C
  manipulate integers of any size (memory permitting)

Finite Precision: Real Representation

- commonly 8 bytes used to store a **double** variable
- 8 bytes $\rightarrow$ 64 bits $\rightarrow$ 2^{64} possible values (bit patterns)
- 64-bits gives huge number of patterns

- use of bit patterns more complex, if you want to know now `https://en.wikipedia.org/wiki/Double-precision_floating-point_format`
- reals in (absolute) range 10^{-308} to 10^{308} can be approximated
- approximation errors can accumulate
- More later and in COMP1521

Numbers and Types

- Numbers in programs have types.
- Numbers with a decimal point are type **double**, e.g.
  3.14159 -34.56 42.0
- C also lets write numbers in scientific notation:
  $2.4e5 \implies 2.4 \times 10^5 \implies 240000.0$
  Numbers in scientific notation are also type **double**
- Numbers without decimal point or exponent are type **int**, e.g.
  42 0 -24
- Numbers in programs are often called constants
  (unlike variables they don't change)

Mathematical functions

- Mathematical functions not part of standard library
  Essentially because tiny CPUs may not support them
- A library of mathematical functions is available including:
  sqrt(), sin(), cos(), log(), exp()
  Above functions take a **double** as argument and return a **double**
- Functions covered fully later in course
- Extra include line needed at top of program:
  #include <math.h>
  (explained later in course)
- dcc includes maths library by default
  most compilers need extra option:
  gcc needs **-lm** e.g.:

```
gcc -o heron heron.c -lm
```