CS 137 Week 6

ASCII, Characters, Strings and Unicode

October 23rd, 2017

Characters

- Syntax `char c;`
- We've already seen this briefly earlier in the term.
- In C, this is an 8-bit integer.
- The integer can be a code representing printable and unprintable characters.
- Can also store single letters via say `char c='a';`

ASCII

- **American Standard Code for Information Interchange**.
- Uses 7 bits with the 8th bit being either used for a parity check bit or extended ASCII.
- Ranges from 0000000-1111111.
- Image on next slide is courtesy of `http://www.hobbyprojects.com/ascii-table/ascii-table.html`

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Highlights

- Characters 0-31 are control characters
- Characters 48-57 are the numbers 0 to 9
- Characters 65-90 are the letters A to Z
- Characters 97-122 are the letters a to z
- Note that 'A' and 'a' are 32 letters away

Programming With Characters

```
#include <stdio.h>
int main(void) {
  char c = 'a'; //97
  int i = 'a'; //97
  c = 65;
  c += 2; //c = 'C'
  c += 32; //c = 'c'
  c = '\n';
  c = '\0';
  c = '0';
  c += 9;
  return 0;
}
```

Final Comments

- For the English language, ASCII turns out to be enough for most applications.
- However, many languages have far more complicated letter systems and a new way to represent these would be required.
- In order to account for other languages, we now have Unicode which we will discuss in a few lectures.

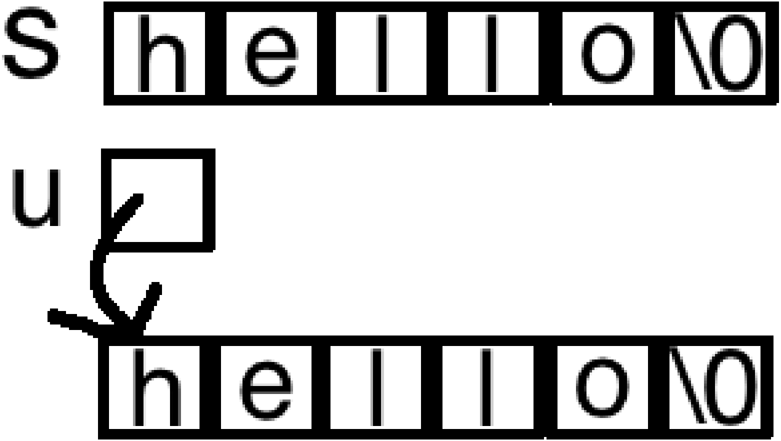
Strings

- In C, strings are arrays of characters terminated by a null character ('\0')

```
#include <stdio.h>
int main(void) {
  char s[] = "Hello";
  printf("%s\n",s);
  //The next is the same as the previous.
  char t[] = {'H','e','l','l','o','\0'};
  printf("%s\n",t);
  //Slightly different
  char *u = "Hello";
  printf("%s\n",u);
  return 0;
}
```

Notice that the last one is slightly different than the previous two...

Slight Change



This Doesn't Seem Like Much But...

```
#include <stdio.h>
int main(void) {
  char s[] = "Hello";
  s[1] = 'a';
  printf("%s\n",s);
  //Slightly different
  char *u = "Hello";
  //The next line causes an error!
  //u[1] = 'a'
  printf("%s\n",u);
  return 0;
}
```

String Literals

- In `char *u = "Hello";`, "Hello" is called a **string literal**.
- String literals are not allowed to be changed and attempting to change them causes undefined behaviour.
- Reminder: Notice also that `sizeof(u)` is different if `u` is an array vs a pointer.
- Another note: `char *hi = "Hello"" world!";` will combine into one string literal.

Question

Write a function that counts the number of times a character c occurs in a string s.

Strings in C

- In C string manipulations are very tedious and cumbersome.
- However, there is a library that can help with some of the basics.
- This being said, there are other languages that are far better at handling string manipulations than C.
- Before discussing these, we need a brief digression into const type qualifiers.

Const Type Qualifiers

- The keyword `const` indicates that something is not modifiable ie. is read-only.
- Assignment to a const object results in an error.
- Useful to tell other programmers about the nature of a variable
- Could tell engineers to store values in ROM.

Examples

- `const int i = 10;` is a constant `i` whose value is initialized to be 10.
- The command `i = 5;` will cause an error because you are trying to reassign a constant.
- Even though it is a constant - through hacks, you could still change the value:

```
#include <stdio.h>
int main(void) {
  const int i = 10;
  printf("%d\n",i);
  int *a = &i;
  *a = 3;
  printf("%d\n",i);
  return 0;
}
```

Differences Between `const` and `#define`

Constants

- `const` can be used to create read-only objects of any time we want, including arrays, structures, pointers etc.
- Constants are subject to the same scope rules as variables
- Constants have memory addresses.

Macros

- `#define` can only be used for numerical, character or string constants.
- Constants created with `#define` aren't subject to the same scoping rules as variables - they apply everywhere.
- Macros don't have addresses.

Important Difference

- The lines
  - `const int *p`
  - `int *const q`

  are very different. The line `const int *p` means that we cannot modify the value that the pointer `p` points to.
- For example, the line `p = &i` is okay whereas the line `*p = 5` will cause an error.
- Continuing on this thought, if we have another pointer `int *r`, then `r = p` will give a warning where as `r = (int *)p` will give no warning but is dubious and in fact `*r = 5` will execute somewhat bypassing the intended behaviour.
- The line `int *const q` means that we cannot modify the actual pointer `q`.
- For example, the line `q = p` will cause an error.

Returning to Strings

- As mentioned before, C has a library to handle strings, `<string.h>` but it contains fairly basic commands when compared to a language like Python.
- Usage:

```
#include <stdio.h>
int main(void) {
  char str1[10] = "abc", str2[10]="abc";
  if(str1 == str2) printf("Happy!");
  else printf("Sad.");
  return 0;
}
```

Comparing strings will always fail (unless they are pointers to the same string!) We probably don't want this behaviour. Thankfully equality is one of the functions inside the library.

Commands

Some commands of note:

- `size_t strlen(const char *s);`
- `char *strcopy(char *s0, const char *s1)`
- `char *strncopy(char *s0, const char *s1,size_t n)`
- `char *strcat(char *s0, const char *s1);`
- `char *strncat(char *s0, const char *s1, size_t n);`
- `int strcmp(const char *s0, const char *s1);`

strlen

`size_t strlen(const char *s);`

- Returns the string length of `s`.
- Does not include the null character.
- Here, the keyword `const` means that strlen should only read the string and not mutate it.

strcopy

`char *strcopy(char *s0, const char *s1)`

- Copies the string `s1` into `s0` and returns `s0`
- `s0` must have enough room to store the contents of `s1` but this check is **not** done inside this function.
- If there is not enough room, `strcopy` will overwrite bits that follow `s0` which is extremely undesirable.
- Why return a pointer? Makes it easier to nest the call if needed.

`char *strncopy(char *s0, const char *s1,size_t n)`

- Only copies the first n characters from `s1` to `s0`.

strcat

`char *strcat(char *s0, const char *s1);`

- Concatenates `s1` to `s0` and returns `s0`
- Does not check if there is enough room in `s0` like `strcopy`.

`char *strncat(char *s0, const char *s1, size_t n);`

- Only concatenates the first *n* characters fro `s1` to `s0`.

`strcmp`

```
int strcmp(const char *s0, const char *s1);
```

- Compares the two strings lexicographically (ie. comparing ASCII values).
- Return value is
  - < 0 if `s0 < s1`
  - > 0 if `s0 > s1`
  - $= 0$ if `s0 == s1`

Examples

```
#include <stdio.h>
#include <string.h>
int main(void){
  char s[10] = "apples";
  char t[] = " to monkeys";
  char u[10];
  strcpy(u,s);
  strncat(s,t,4);
  strcat(s,u);
  printf("%s\n",s);
  int comp = strcmp("abc","azenew");
  //Remember if s0 < s1 <-> comp < 0
  if (comp < 0) printf("value is %d\n",comp);
  comp = strcmp("ZZZ","a");
  if (comp < 0) printf("value is %d\n",comp);
}
```

Exercise

- Notice that strcat modifies the first string.
- Write a program that concatenates two strings into a new string variable and returns a pointer to this object.

`gets` vs `scanf`

- Very briefly, when trying to read a string from the user using scanf, recall that it stops reading characters at any whitespace type character
- This might not be the desired effect - to change this, you could use the `gets` function which stops reading input on a newline character.
- Both are risky functions as they don't check to see when the array which is storing the strings are full.
- Often C programmers will just write their own input functions to be safe.

Printing Strings

- On certain compilers, eg `gcc -std=c11`, the command

  ```
  char *s = "abcj\n"; printf(s);
  ```

  gives a warning that this is not a string literal and no format arguments.
- Turns out this is a potential security issue if the string itself contains formatting arguments (for example if it was user created)
- You can avoid these errors if for example you make the above string a constant or if you use `printf("%s",s);` type commands.

Other String Functions

- `void *memcopy(void * restrict s1, const void *restrict s2, size_t n)`
- `void *memmove(void *s1, const void * s2, size_t n)`
- `int *memcmp(const void *s1, const void *s2, size_t n)`
- `void *memset(void *s, int c, size_t n)`

memcopy

`void *memcopy(void * restrict s1, const void *restrict s2, size_t n)`

- Copies n bytes from `s2` to `s1` which must not overlap.
- `restrict` indicates that only this pointer will access that memory area. This allows for compiler optimizations.
- For example

```
#include <stdio.h>
#include <string.h>
int main(void) {
  char s[10];
  memcpy(s, "hello",5);
  printf("%s\n",s);
  return 0;
}
```

memmove

`void *memmove(void *s1, const void * s2, size_t n)`

- Similar to `memcopy` but `s1` and `s2` can overlap.
- For example

```c
#include <stdio.h>
#include <string.h>
int main (void) {
    char dest[] = "oldvalue";
    char src[]  = "newvalue";
    printf("Pre memmove,
      dest: %s, src: %s\n", dest, src);
    memmove(dest, src, 9);
    printf("Post memmove,
      dest: %s, src: %s\n", dest, src);
    return(0);
}
```

memcmp

`int *memcmp(const void *s1, const void *s2, size_t n)`

- Similar to strcmp except it compares the bytes of memory.
- For example,

```
#include <stdio.h>
int main(void) {
  char s[10] = "abc";
  char t[10] = "abd";
  int val = memcmp(s,t,2);
  if (val ==0) printf("Amazing!");
  return 0;
}
```

memset

`void *memset(void *s, int c, size_t n)`

- Fills the first `n` bytes of area with byte `c`.
- For example

```
#include <stdio.h>
int main(void) {
  int a[100];
  memset(a,0,sizeof(a));
  printf("%d\n",a[43]);
  memset(a,1,sizeof(a));
  printf("%d\n",a[41]);
  //1 + 2^{8} + 2^{16} +2^{24} = 16843009
  return 0;
}
```

Unicode

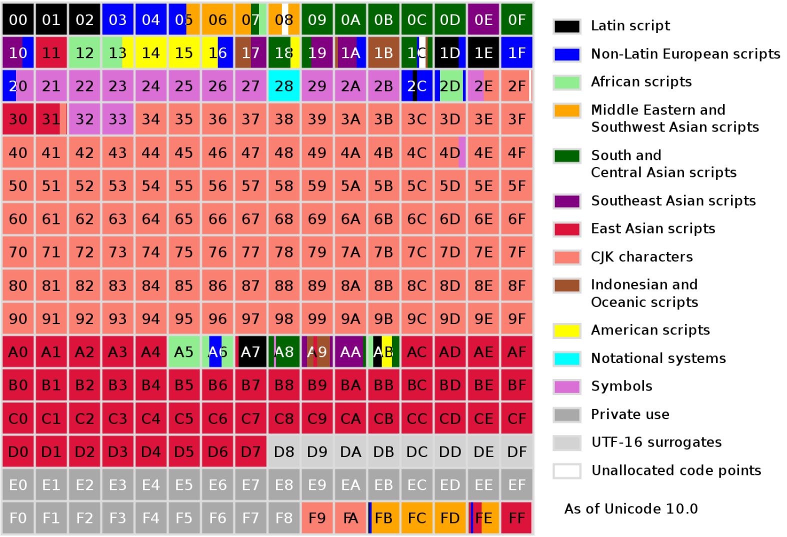
- As exciting as ASCII is, it is far from sufficient to handle all characters over all languages/alphabets.
- Unicode spans more than 100,000 characters over languages both real and fake, both living and dead!
- A unicode character spans 21 bits and has a range of 0 to $1,114,112$ or 3 bytes per character. This last number comes from the 17 planes which unicode is divided into multiplied by the 2^{16} code points (contiguous block).
- Plane 0 is the BMP (Basic Multilingual Plane) - see next slide.
- Unicode letters also share the same values as ASCII. This was necessary for adoption by the Western World which had ASCII first.
- Examples:

UTF+13079

UTF+0061 $(6 \cdot 16 + 1 = 97)$

a

First Plane Basic Multilingual Plane



More on Unicode Planes

- Plane 0 (BMP) consists of characters from U+0000 to U+FFFF
- Plane 1 consists of characters from U+10000 to U+1FFFF
- ... Plane 15 consists of characters from U+F0000 to U+FFFFF
- Plane 16 consists of characters from U+100000 to U+10FFFF

Unicode Encoding

- The Unicode specification just defines a character code for each letter.
- There are different ways however to actually **encode** unicode.
- Popular encodings include UTF-8, UTF-16, UTF-32, UCS-2.
- Different encodings have advantages and disadvantages
- We'll talk about UTF-8, one of the best supported encodings.

Byte Usage in UTF-8

Code Point Range in Hex	UTF-8 Byte Sequence in Binary
000000-00007F	0xxxxxxx
000080-0007FF	110xxxxx 10xxxxxx
000800-00FFFF	1110xxxx 10xxxxxx 10xxxxxx
010000-01FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

- For example, let's look at the letter ä which has unicode 0xE4 or 11100100 in binary.
- In UTF-8, this falls into range 2 above and so is encoded as 110**00011** 10**1000100**.
- When you concatenate the bolded text gives you the binary encoding of 0xE4.

More on Byte Usage in UTF-8

- The 1 byte characters (ie those in range 1) correspond to the ASCII characters (0 to 0x7F $=$ 01111 1111 $=$ 127)
- The 2 byte characters are up to 11 bits long with a range 128 to $2^{11} - 1$ (ie 2047)
- The 3 byte characters are up to 16 bits long, with a range 2048 to $2^{16} - 1$ (ie 65535)
- The 4 byte characters are up to 21 bits long, with a range 65536 to $2^{21} - 1$ (ie: 2097151)

Notes

- In C, a standard library called `<wchar.h>` has code for dealing with unicode.
- In fact, more popularly, ICU (the International Components for Unicode) is more in use by companies such as Adobe, Amazon, Appache, Apple, Google, IBM, Oracle, etc.
- For more details, visit `http://site.icu-project.org/`
- For us we will mainly be dealing with ASCII.
- However, in an ever international world, you will need to at some point understand Unicode encoding.

Example using <wchar.h>

```
#include <locale.h>
#include <wchar.h>
int main(void) {
  wchar_t wc = L’x3b1’;
  setlocale(LC_ALL, "en_US.UTF-8");
  wprintf(L"%lc\n",wc);
  wprintf(L"%zu\n",sizeof(wchar_t));
  return 0;
}
```

This Week

- We spoke a lot about characters and strings, including how they are encoded and how to program them in C
- Next week we make a big shift to discuss algorithm efficiency.
- We will discuss Big-Oh Notation and it's relatives and then use the notation to discuss many sorting algorithms.