CS 137 Week 6
ASCII, Characters, Strings and Unicode

October 23rd, 2017
• 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';`
• **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.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Start of heading</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Start of text</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>End of text</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>End of transmit</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>Enquiry</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>Acknowledge</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>Audible bell</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>Backspace</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>Horizontal tab</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Form feed</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Vertical tab</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Carriage return</td>
<td>45</td>
<td>2D</td>
<td>_</td>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Shift out</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Shift in</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Data link escape</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Device control 1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Neg. acknowledge</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Synchronous idle</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>End trans. block</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>Cancel</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>End of medium</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>Substitution</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Escape</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>File separator</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>Group separator</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Record separator</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Unit separator</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>`</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>96</td>
<td>60</td>
<td>'</td>
</tr>
</tbody>
</table>
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
#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 = '
';
    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')

```c
#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

s

hello\0

u

t

hello\0
#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.
The keyword `const` indicates that something is not modifiable, i.e., 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:

```c
#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:

```c
#include <stdio.h>
int main(void) {
    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 *strncpy(char *s0, const char *s1)`
- `char *strncpy(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.
strcpy

char *strcpy(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, strcpy 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 strncpy.

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

- Only concatenates the first n characters from s1 to s0.
int strcmp(const char *s0, const char *s1);

- Compares the two strings lexicographically (i.e., comparing ASCII values).
- Return value is
  - < 0 if s0 < s1
  - > 0 if s0 > s1
  - = 0 if s0 == s1


```c
#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","azene\n");
    //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
  
  ```c
  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 *memcpy(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)`
memcpy

void *memcpy(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

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

- Similar to `memcpy` 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**

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

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

```c
#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

```c
#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 ($6 \cdot 16 + 1 = 97$)
# First Plane Basic Multilingual Plane

As of Unicode 10.0
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 $U+10FFFF$
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.
<table>
<thead>
<tr>
<th>Code Point Range in Hex</th>
<th>UTF-8 Byte Sequence in Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000-00007F</td>
<td>0xxxxxxxx</td>
</tr>
<tr>
<td>000080-0007FF</td>
<td>110xxxxx 10xxxxxxx</td>
</tr>
<tr>
<td>000800-00FFFF</td>
<td>1110xxxx 10xxxxxxx 10xxxxxxx</td>
</tr>
<tr>
<td>010000-01FFFF</td>
<td>11110xxx 10xxxxxxx 10xxxxxxx 10xxxxxx</td>
</tr>
</tbody>
</table>

- 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 11000011 101000100.
- 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)
• 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>`

```c
#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 its relatives and then use the notation to discuss many sorting algorithms.