CS 33

Introduction to C
Part 5
Basic Data Types

- **int**: -2,147,483,648 – 2,147,483,647
- **short**: -32,768 – 32,767
- **long (on 64-bit computer)**: -9,223,372,036,854,775,808 – 9,223,372,036,854,775,807
- **float**: ~10e-44.85 – ~10e38.53, 23-bit significand
- **double**: ~10e-323.3 – ~10e308.3, 52-bit significand
- **char**: -128 – 127
Characters

• ASCII
  – American Standard Code for Information Interchange
  – works for:
    » English
    » Swahili
  – doesn’t work for:
    » French
    » Dutch
    » Spanish
    » German
    » Arabic
    » Sanskrit
    » Chinese
    » pretty much everything else
Who cares!!
You should care …

(but not in this course)
# ASCII Character Set

<table>
<thead>
<tr>
<th>00</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>110</th>
<th>120</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: \0 \n</td>
<td>2</td>
<td>&lt;</td>
<td>F</td>
<td>P</td>
<td>Z</td>
<td>d</td>
<td>n</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: \v</td>
<td>3</td>
<td>=</td>
<td>G</td>
<td>Q</td>
<td>[</td>
<td>e</td>
<td>o</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: \f</td>
<td>sp</td>
<td>4</td>
<td>&gt;</td>
<td>H</td>
<td>R</td>
<td>\</td>
<td>f</td>
<td>p</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: \r</td>
<td>!</td>
<td>+</td>
<td>5</td>
<td>?</td>
<td>I</td>
<td>S</td>
<td>]</td>
<td>g</td>
<td>q</td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: &quot;</td>
<td>,</td>
<td>6</td>
<td>@</td>
<td>J</td>
<td>T</td>
<td>^</td>
<td>h</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: #</td>
<td>-</td>
<td>7</td>
<td>A</td>
<td>K</td>
<td>U</td>
<td>_</td>
<td>i</td>
<td>s</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: $</td>
<td>.</td>
<td>8</td>
<td>B</td>
<td>L</td>
<td>V</td>
<td>`</td>
<td>j</td>
<td>t</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: \a</td>
<td>%</td>
<td>/</td>
<td>9</td>
<td>C</td>
<td>M</td>
<td>W</td>
<td>a</td>
<td>k</td>
<td>u</td>
<td>DEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8: \b</td>
<td>&amp;</td>
<td>0</td>
<td>:</td>
<td>D</td>
<td>N</td>
<td>X</td>
<td>b</td>
<td>l</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9: \t</td>
<td>'</td>
<td>1</td>
<td>;</td>
<td>E</td>
<td>O</td>
<td>Y</td>
<td>c</td>
<td>m</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
chars as Integers

```c
char tolower(char c) {
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```
Character Strings

```c
char c = 'a';
```

```c
char *s = "string";
```

Is there any difference between \texttt{c1} and \texttt{c2} in the following?

\begin{verbatim}
char c1 = 'a';
char *c2 = "a";
\end{verbatim}
Yes!!

```c
char c1 = 'a';

char *c2 = "a";
```

- `c1`: a
- `c2`: a \0
What do \( s1 \) and \( s2 \) refer to after the following is executed?

```c
char s1[] = "abcd";
char *s2 = s1;
s1[0] = 'z';
s2[2] = '\0';
```
Weird ...

Suppose we did it this way:

```c
char *s1 = "abcd";
char *s2 = s1;
s1[0] = 'z';
s1[2] = '\0';
```

```bash
% gcc -o char char.c
% ./char
Segmentation fault
```
Copying Strings (1)

```c
char s1[] = "abcd";
char s2[5];

s2 = s1;    // does this do anything useful?

// correct code for copying a string
for (i=0; s1[i] != '\0'; i++)
    s2[i] = s1[i];
s2[i] = '\0';

// would it work if s2 were declared:
char *s2;
// ?
```
Copying Strings (2)

```c
char s1[] = "abcdefghijklmnopqrstuvwxyz";
char s2[5];

for (i=0; s1[i] != '\0'; i++)
    s2[i] = s1[i];
s2[i] = '\0';

Does this work?

for (i=0; (i<4) && (s1[i] != '\0'); i++)
    s2[i] = s1[i];
s2[i] = '\0';
```
String Length

```c
char *s1;

s1 = produce_a_string();
// how long is the string?

sizeof(s1); // doesn’t yield the length!!

for (i=0; s1[i] != '\0'; i++)
;
// number of characters in s1 is i
```
int main() {
    char s[] = "1234";
    printf("%d\n", sizeof(s));
    proc(s, 5);
    return 0;
}

void proc(char s1[], int len) {
    char s2[12];
    printf("%d\n", sizeof(s1));
    printf("%d\n", sizeof(s2));
}

$ gcc -o size size.c
$ ./size
5
8
12
$
void proc(char s[16]) {
    printf("%d\n", sizeof(s));
}

What’s printed?

a) 8  

b) 15 

c) 16 

d) 17
Comparing Strings (1)

```c
char *s1;
char *s2;

s1 = produce_a_string();
s2 = produce_another_string();
// how can we tell if the strings are the same?

if (s1 == s2) {
    // does this mean the strings are the same?
} else {
    // does this mean the strings are different?
}
```
Comparing Strings (2)

```c
int strcmp(char *s1, char *s2) {
    int i;
    for (i=0;
         (s1[i] == s2[i]) && (s1[i] != 0) && (s2[i] != 0);
         i++)
        ; // an empty statement
    if (s1[i] == 0) {
        if (s2[i] == 0) return 0; // strings are identical
        else return -1; // s1 < s2
    } else if (s2[i] == 0) return 1; // s2 < s1
    if (s1[i] < s2[i]) return -1; // s1 < s2
    else return 1; // s2 < s1;
}
```
The String Library

#include <string.h>

char *strcpy(char *dest, char *src);
    // copy src to dest, returns ptr to dest
char *strncpy(char *dest, char *src, int n);
    // copy at most n bytes from src to dest
int strlen(char *s);
    // return the length of s (not counting the null)
int strcmp(char *s1, char *s2);
    // returns -1, 0, or 1 depending on whether s1 is
    // less than, the same as, or greater than s2
int strncmp(char *s1, char *s2, int n);
    // do the same, but for at most n bytes
The String Library (more)

```c
size_t strspn(const char *s, const char *accept);
    // returns length of initial portion of s
    // consisting entirely of bytes from accept

size_t strcspn(const char *s, const char *reject);
    // returns length of initial portion of s
    // consisting entirely of bytes not from
    // reject
```
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "Hello World!\n";
    char *s2;
    strcpy(s2, s1);
    printf("%s", s2);
    return 0;
}

This code:

a) is a great example of well written C code
b) has syntax problems
c) might seg fault
Parsing a String

\[
\begin{array}{cccccc}
\text{arg1} & \text{arg2} & \backslash 0 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
\text{arg1} & \backslash 0 & \text{arg2} & \backslash 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{arg1} & \backslash 0 & \text{arg2} & \backslash 0 & \backslash 0 \\
\end{array}
\]
Design of *getfirstword*

- char *getfirstword(char **rem_p)*
  - returns
    » pointer to null-terminated first word in *rem_p*
    or
    » NULL, if *rem_p* is a string of entirely whitespace
  - *rem_p* modified to
    » point to character following first word in *rem_p* if within bounds of string
    or
    » NULL if next character not within bounds
Using `getfirstword`

```c
int main() {
    char line[] = " arg0 arg1 arg2 arg3 ";
    char *rem = line;
    char *str;
    while ((str = getfirstword(&rem)) != NULL) {
        printf("%s\n", str);
    }
    return 0;
}
```

Output:
arg0
arg1
arg2
arg3
char *getfirstword(char **rem_p) {
  char *str = *rem_p;
  if (str == NULL)
    return NULL;
  int len = strlen(str);
  int wslen =
    strspn(str, " 	\n");
    // initial whitespace
  if (wslen == len) {
    // string is all whitespace
    return NULL;
  }
  str = &str[wslen];
  // skip over whitespace
  len -= wslen;

  int wlen =
    strcspn(str, " \t\n");
    // length of first word
  if (wlen < len) {
    // word ends before end of
    // string: terminate
    str[wlen] = '\0';
    *rem_p = &str[wlen+1];
  } else {
    // no more words
    *rem_p = NULL;
  }
  return str;
}