Library Strings Functions

Based on slides from K. N. King and Dianna Xu

Bryn Mawr College
CS246 Programming Paradigm

Using the C String Library

- Strings are treated as arrays in C.
- Strings can't be copied or compared using operators.

```
char str1[10], str2[10];
...
str1 = "abc";  /*** WRONG ***/
str2 = str1;   /*** WRONG ***/
if(str2 == str1) /*** WRONG ***/
```

Since str1 and str2 have different addresses, the expression `str1 == str2` must have the value 0.

Using an array name as the left operand of = is illegal.

- *Initializing* a character array using = is legal, though:

```
char str1[10] = "abc";
```

In this context, = is not the assignment operator.

Library String Functions

- `#include <string.h>`
- Find the Length of a string

```
size_t strlen(const char *str)
```

- Copy a string (including the '\0')

```
char *strcpy(char *t, const char *s)
```

- Concatenate two strings

```
char *strcat(char *t, const char *s)
```

- Compare two strings

```
int strcmp(const char *s1, const char *s2)
```

Return: 0 if identical; ASCII difference between the first mismatch otherwise

"abc" vs. "abb" : +1; "abc" vs. "abd" : -1

Example

```
int main() {
  char s[] = "ann"; char s2[] = "abby";
  char s3[strlen(s)+strlen(s2)+1];

  printf("%d\n", strlen(s));
  printf("%d\n", strlen(&s[1]));

  strcpy(s3, s);
  strcat(s3, s2);
  printf("%s\n", &s3[2]);

  printf("%d\n", strcmp(s, s2));

  return 0;
}
```

Length Function `strlen`

```
size_t strlen(const char *s) {
  size_t n;
  for (n = 0; *s != '\0'; s++)
    n++;
  return n;
}
```

Length Function `strlen`

```
size_t strlen(const char *s){
  size_t n = 0;
  for (; *s != '\0'; s++)
    n++;
  return n;
}
```

Length Function **strlen**

```
size_t strlen(const char *s) {
  size_t n = 0;
  for (; *s; s++)
    n++;
  return n;
}
```

Length Function **strlen**

```
size_t strlen(const char *s) {
  size_t n = 0;
  for (; *s++; )
    n++;
  return n;
}
```

Length Function **strlen**

```
size_t strlen(const char *s) {
  size_t n = 0;
  while (*s++)
    n++;
  return n;
}
```

Length Function **strlen**

```
size_t strlen(const char *s) {
  const char *p = s;
  while (*s)
    s++;
  return s - p;
}
```

Searching for the End of a String

```
while (*s)      while (*s++)
  s++;              ;
```

- The first version leaves `s` pointing to the null character.
- The second version is more concise, but leaves `s` pointing just past the null character.

Copy Function **strcpy**

```
char* strcpy(char to[], char from[]) {
  int i;
  for (i = 0; from[i] != '\0'; i++)
    to[i] = from[i];
  to[i] = '\0'; return to;
}
```

```
char* strcpy(char *to, char *from) {
  char *tmp = to;
  while ((*to = *from)!= '\0'){
    to++; from++;
  }
  return tmp;}
```

strcpy, pointer version

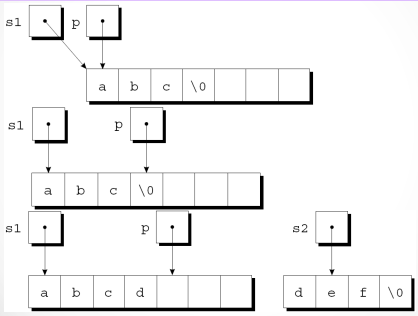
```
char* strcpy(char *to, char *from) {
  char *tmp = to;
  while ((*to++ = *from++)!= '\0');

  return tmp;
}
```

```
char* strcpy(char *to, char *from) {
  char *tmp = to;
  while (*to++ = *from++);

  return tmp;
}
```

strcat

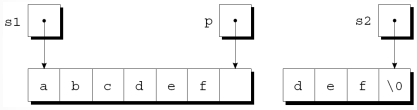
```
char *strcat(char *s1, const char *s2){
  char *p = s1;

  while (*p != '\0')
    p++;
  while (*s2 != '\0') {
    *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';
  return s1;
}
```

strcat



strcat

- The loop terminates when `s2` points to the null character:



- After putting a null character where `p` is pointing, `strcat` returns.

Condensed version strcat

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p)
    p++;
  while (*p++ = *s2++)
    ;
  return s1;
```

- What causes the loop to terminate?

Comparison strcmp

```
int strcmp(char *s, char *t) {

  for(;*s == *t; s++,t++) {
    if (*s == '\0')
      return 0;
  }

  return *s-*t;
}
```

Cannot compare strings using **==**

- Returns **0** if **s == t**
- If not, returns the difference btw the first chars that differ

Other Library String Functions

- n functions

```
char *strncpy(char *t, const char *s, size_t n)
char *strncat(char *t, const char *s, size_t n)
int strncmp(const char *s1, const char *s2, size_t n)
```

  - Same as the none-n functions, only works on n chars
- A safer way to use `strncpy`:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

- Search for a character in a string

```
char* strchr(char *s, char c)
```

- Search for a (sub)string in a string

```
char* strstr(char *s, char *substr)
```

`string.h` Functions Example

```
int main() {
  char s[] = "abcdejklijkl", s2 = 'b';
  char *s3 = "jkl", *s4, *s5;

  s4 = strchr(s, s2);
  s5 = strstr(s, s3);

  printf("%s\n", s4);
  printf("%c\n", s5);       //?
  printf("%d\n", s5-s4);
  printf("%s\n", &(s[2]));
}
```

`string.h` Functions Example

```
int main() {
  char s[] = "abcdefghijkl", s2 = 'g';
  char *s3 = "jkl", *s4, *s5, *s6;

  s4 = strchr(s, s2);
  s5 = strstr(s, s3);
  s6 = s;

  printf("%s\n", s6+6);
  printf("%c\n", *(s6++));
  printf("%c\n", *(++s4));
  printf("%c\n", ++(*s4));
  printf("%d\n", s5-s4);
  printf("%s\n", &(s6[2]));
}
```

Using library functions

```
int main() {
  char s1[] = "The name is Bond";
  char s2[] = "Bond, James Bond";
  char s3[100];

  strncpy(s3, s1, 12);
  s3[12] = '\0';
  strncat(s3, &s2[6], 5);

  printf("%s\n", s3);
}
```

- Remember to always null-terminate a string
- `string.h` functions may have undefined behaviors otherwise

Arrays of Strings

- Array of strings ==> two-dimensional character array, with one string per row:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

- The number of rows in the array can be omitted, but we must specify the number of columns.

Arrays of Strings

- Unfortunately, the `planets` array contains a fair bit of wasted space (extra null characters):

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

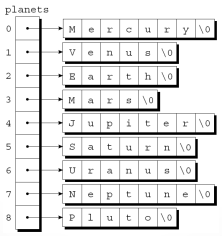
Arrays of Strings

- Most collections of strings will have a mixture of long strings and short strings.
- In a ragged array, rows can have different lengths.
- We can simulate a ragged array in C by creating an array whose elements are pointers to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

Arrays of Strings

- This small change has a dramatic effect on how `planets` is stored:



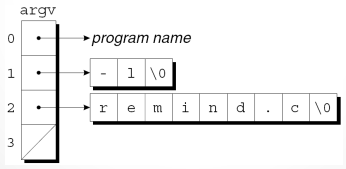
Arrays of Strings

- To access one of the planet names, all we need do is subscript the `planets` array.
- A loop that searches the `planets` array for strings beginning with the letter M:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```

Command-Line Arguments

- Available through two parameters to **`main`**
  - **`main(int argc, char *argv[])`**
  - **`argc`** – argument count, i.e. number of args
  - **`argv`** – an array of pointers to the arguments
- **`ls -l remind.c`**



Command-Line Arguments

- Use an integer variable as an index into the `argv` array to access command-line arguments:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

- Set up a pointer to `argv[1]`, then increment the pointer repeatedly:

```
char **p;

for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```