String Processing in C

C Programming and Software Tools

N.C. State Department of Computer Science
Standard Library: `<ctype.h>`

- Many functions for **checking** whether a character is a digit, is upper case, ...
  - `isalnum(c), isalpha(c), isspace(c),`

- Also, functions for **converting** to upper case and converting to lower case
  - `toupper(c), tolower(c),`

- Argument is an `int` and return is an `int`
  - Works fine with unsigned chars or 7-bit character types
  - Need to cast to `unsigned char` for safety
### <ctype.h> (cont’d)

#### Checking:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum (c)</td>
<td>c is a letter or a digit</td>
</tr>
<tr>
<td>isalpha (c)</td>
<td>c is a letter</td>
</tr>
<tr>
<td>isdigit (c)</td>
<td>c is a decimal digit</td>
</tr>
<tr>
<td>islower (c)</td>
<td>c is a lower-case letter</td>
</tr>
<tr>
<td>isspace (c)</td>
<td>c is white space (\f\n\r\t\v)</td>
</tr>
<tr>
<td>isupper (c)</td>
<td>c is an upper-case letter</td>
</tr>
</tbody>
</table>

#### Converting:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tolower (c)</td>
<td>convert c to lower case</td>
</tr>
<tr>
<td>toupper (c)</td>
<td>convert c to upper case</td>
</tr>
</tbody>
</table>

Only a partial list (see p. 612-613 or library for full list)
Strings

• Simply 1-D arrays of type char, terminated by null character ( ' \0 ' )

• A variety of standard library functions provided for processing
**scanf() and printf() for Strings**

- `sscanf(s, "...", ...)`: scans a string (instead of stdin) for expected input.
- `sprintf(s, "...", ...)`: outputs to a string (instead of stdout) the specified output.
Standard Library: `<string.h>`

- Lots of string processing functions for
  - copying one string to another
  - comparing two strings
  - determining the length of a string
  - concatenating two strings
  - finding a substring in another string
  - ...

- Function headers at end of slides
- More details in King text book (Section 23.6)
A Useful Memory Operation: `memcpy()`

- Must `#include <string.h>`
- Syntax:
  ```c
  void * memcpy (void * dest, void * src, size_t n)
  ```
- Copy `n` bytes from memory pointed to by `src` to memory pointed to by `dest`
  - memory areas must not overlap!
- Returns pointer to `dest`
memcpy() (cont’d)

• Since C does not have an operator to assign one array to another, this is a handy function

```c
#define SZ 1000
int *ip, *jp;

int A[1000], B[1000];

... assign some values to A ...

memcpy(B, A, 1000*sizeof(int));
```
Variant: \textit{memmove()} 

- \textit{memmove()} works just like \textit{memcpy()}, except \texttt{src} and \texttt{dest} areas may overlap.
Another Useful Operation:

**memcmp()**

- Syntax:
  ```
  int memcmp (void *s1, void *s2, size_t n)
  ```
- Returns 0 if `n` bytes starting at `s1` are equal to `n` bytes starting at `s2`
- Else, return `val < 0` if first non-equal byte of `s1` < byte of `s2` ,  > 0 if ...
- Useful for comparing arrays, but byte-by-byte comparison only
  - e.g., don't use for comparing arrays of ints, floats, structs, etc.
**memcmp() ... (cont'd)**

```c
char X[1000], Y[1000];
int A[1000], B[1000];

... assign some values to A, B, X, Y ...

if (memcmp(X, Y, 1000) < 0)
    ...X is less than Y...
```

Do not try this as-is with A and B; why not?
# String function summary

<table>
<thead>
<tr>
<th>Raw memory</th>
<th>String</th>
<th>String with limit</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcpy</td>
<td>strcpy</td>
<td>strncpy</td>
<td>Copy</td>
</tr>
<tr>
<td>memmove</td>
<td>-</td>
<td>strncat</td>
<td>Concatenate (append) strings</td>
</tr>
<tr>
<td>strcmp</td>
<td>strcmp</td>
<td>strncmp</td>
<td>Compare</td>
</tr>
<tr>
<td>memchr</td>
<td>strchr</td>
<td>-</td>
<td>Find a char</td>
</tr>
<tr>
<td>strstr</td>
<td>-</td>
<td>strpbrk</td>
<td>Find any of a set of chars in a string</td>
</tr>
<tr>
<td>strtok</td>
<td>-</td>
<td>strlen</td>
<td>Find the length of a string</td>
</tr>
<tr>
<td>strspn</td>
<td>strcspn</td>
<td>strpbrk</td>
<td>Find any of a set of chars in a string</td>
</tr>
</tbody>
</table>


1 Allows overlapping memory
2 Reverse (right-to-left) search
Danger zone (1)

- What’s wrong with this?

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    char filename[32];
    strcpy(argv[1], filename);
    printf("Opening %s...\n", filename);
    // more code goes here
    return 0;
}
```
Danger zone (2)

- What’s wrong with this, then?

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    char filename[32];
    strcpy(filename, argv[1]);
    printf("Opening %s...\n", filename);
    // more code goes here
    return 0;
}
```
The common way to fix this

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    char* filename;
    filename = argv[1];
    printf("Opening %s...\n", filename);
    // more code goes here
    return 0;
}
```
• If you absolutely need a *copy* of the string.

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    char* filename = malloc(strlen(argv[1])+1);
    strcpy(filename, argv[1]);
    printf("Opening %s...\n", filename);
    // more code goes here
    return 0;
}
```
Good Practice

• You should be able to write the code for any of the standard library functions
  – e.g., computing the length of a string...

```c
char s[1000] = "a string";
char *p = s;

while (*p++)
{
}

return (p - s);
```
<stdlib.h> String Functions

- **double atof( char s[] )** converts a string to a **double**, ignoring leading white space
- **int atoi( char s[] )** converts a string to an **int**, ignoring leading white space
  - These don’t return information about errors
- **Could also use**
  - **strtol**
  - **strtol/f**
  - **sscanf**

That sucks.

Fine, but error reporting is a little complicated.

Nicest, but expensive.
Arrays of Strings

• Creating a two dimensional array of chars is inefficient
  – Wasted space when strings of different lengths
• Instead we want a ragged array
  – Create an array where the elements are pointers to strings

```c
```
Arrays of Strings (con’t)

• Accessing a string in the array
  - `planets[i]`

• Accessing a character in a string
  - `planets[i][j]`

Example:
```
for (int i = 0; i < 8; i++)
    if (planets[i][0] == 'M')
        printf("%s\n", planets[i]);
```
Exercise 15a
Upper-case-ify

• Make a function that does this:

```c
void uppercaseify(char* c) {
    // YOUR CODE HERE
}

int main() {
    char s[] = "Hey everyone!";
    printf("%s\n",s); // Hey everyone!
    uppercaseify(s);
    printf("%s\n",s); // HEY EVERYONE!
}
```

Pro-mode: Don’t use any brackets in uppercaseify
HERE’S A BUNCH OF FUNCTION PROTOTYPES YOU CAN READ YOURSELF

Better yet, read the manpages, or a C reference library like cplusplus.com.
\texttt{<string.h>}: Copying

- \texttt{void *memcpy(void * restrict s1, const void * restrict s2, size_t n);}
- \texttt{void *memmove(void *s1, const void *s2, size_t n);}
- \texttt{char * strcpy(char * restrict s1, const char * restrict s2);}  
- \texttt{char *strncpy(char * restrict s1, const char * restrict s2, size_t n)}
<string.h>: Concatenation

- char *strcat(char * restrict s1, const char * restrict s2);
- char *strncat(char * restrict s1, const char * restrict s2, size_t n);
<string.h>: Comparison

- int memcmp(const void *s1, const void *s2, size_t n);
  - n comparisons
- int strcmp(const char *s1, const char *s2);
  - Stops when reaches null in either string
- int strcoll(const char *s1, const char *s2);
  - Locale dependent
- int strncmp(const char *s1, const char *s2, size_t n);
  - Stops when reaches null in either string or n comparisons, which ever is first
<string.h>: Search

- `void *memchr(const void *s, int c, size_t n);`
  - Like strchr, but stops searching after n characters
- `char *strchr(const char *s, int c);`
  - Searches a string for a particular character
  - Use pointer arithmetic to find additional characters
- `size_t strcspn(const char *s1, const char *s2);`
  - Index of first character that’s in the set s2
- `char *strpbrk(const char *s1, const char *s2);`
  - Pointer to leftmost character in s1 that matches any character in s2
<string.h>: Search

- `char *strrchr(const char *s, int c);`
  - Searches string in reverse order
- `size_t strspn(const char *s1, const char *s2);`
  - Index of first character that’s NOT in the set `s2`
- `char *strstr(const char *s1, const char *s2);`
  - Pointer to first occurrence of `s2` in `s1`
- `char *strtok(char * restrict s1, const char * restrict s2);`
  - Scans `s1` for the non-empty sequence of characters that are not in `s2`
  - Use to tokenize strings
<string.h>: Other Functions

- **void *memset(void *s, int c, size_t n);**
  - Stores copy of c to area of memory of size n

- **size_t strlen(const char *s);**
  - Length of the string, not counting the null character
Command Line Arguments

• To use command line arguments, define main as:

```c
int main(int argc, char *argv[]) {}
```

  • `argc`: argument count
    • Includes the program itself
  • `argv`: argument vector
    • Array of pointers to command line arguments stored as strings
    • `argv[0]`: name of program
    • `argv[1]–argv[argc-1]`: other arguments
    • `argv[argc]`: null pointer
Processing Command Line Args

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

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