57:017, Computers in Engineering—Quick Review of C Character Strings

Introduction

- Introduce some standard library functions
  - Easy string and character processing
  - Programs can process characters, strings, lines of text, and blocks of memory
- These techniques used to make:
  - Word processors
  - Page layout software
  - Typesetting programs
  - etc.

Fundamentals of Strings and Characters

- Characters
  - Character constant
    - represented as a character in single quotes
    - 'z' represents the integer value of z
    - stored in ASCII format (one byte/character)
- Strings
  - Series of characters treated as a single unit
    - Can include letters, digits and special characters (*, /, $)
  - String literal (string constant) - written in double quotes
    - "Hello"
  - Strings are arrays of characters
    - The type of a string is char * i.e. “pointer to char”
    - Value of string is the address of first character

The ASCII Character Set

Strings and Characters

- String declarations
  - Declare as a character array or a variable of type char *
    - char color[] = "blue";
    - char *colorPtr = "blue";
  - Remember that strings represented as character arrays end with '\0'
    - color has 5 elements
- Inputting strings
  - Use scanf
    - char word[10];
    - scanf("%s", word);
  - Copies input into word[]
    - Do not need & (because word is a pointer)
  - Remember to leave room in the array for '\0'
Returns

true
false
c
32 "hexadecimal digit",
31 isxdigit( 'J' ) ? "J is a “ : "J is not a “,
30 "hexadecimal digit",
29 isxdigit( 'F' ) ? "F is a “ : "F is not a “,
28 "According to isxdigit:"
27 printf( "%s
%s%s
%s%s
%s%s
%s%s
%s%s

", "digit or a letter" );
26 isalnum( '#' ) ? "# is a “ : "# is not a “,
25 "digit or a letter",
24 isalnum( '8' ) ? "8 is a “ : "8 is not a “,
23 isalnum( 'A' ) ? "A is a “ : "A is not a “,
21 printf( "%s
%s%s
%s%s
%s%s

", "4 is not a “, "letter" );
20 isalpha( '4' ) ? "4 is a “ :
19 isalpha( 'b' ) ? "b is a “ : "b is not a “,
18 "letter",
17 isalpha( 'A' ) ? "A is a “ : "A is not a “,
16 printf( "%s
%s%s
%s%s
%s%s

", "4 is not a “, "letter" );
15 isalpha( 'b' ) ? "b is a “ : "b is not a “,
14 "letter",
13 isalnum( 'A' ) ? "A is a “ : "A is not a “,
12 printf( "%s
%s%s
%s%s
%s%s

", "4 is not a “, "letter" );
11 isalpha( 'A' ) ? "A is a “ : "A is not a “,
10 printf( "%s
%s%s
%s%s

", "According to isalpha:"
9 isdigit( '#' ) ? "# is a “ :
8 isdigit( '8' ) ? "8 is a “ : "8 is not a “,
7 printf( "%s
%s%s
%s%s

", "According to isdigit:"
6 int main() { printf( "%d
ivar=98  cvar='a'

}
String Conversion Functions

- Conversion functions
  - In `<stdlib.h>` (general utilities library)
  - Convert strings of digits to integer and floating-point values

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double atof( const char *nPtr )</code></td>
<td>Converts the string nPtr to double.</td>
</tr>
<tr>
<td><code>int atoi( const char *nPtr )</code></td>
<td>Converts the string nPtr to int.</td>
</tr>
<tr>
<td><code>long atol( const char *nPtr )</code></td>
<td>Converts the string nPtr to long.</td>
</tr>
<tr>
<td><code>unsigned long strtoul( const char *nPtr, char **endPtr, int base )</code></td>
<td>Converts the string nPtr to unsigned long.</td>
</tr>
<tr>
<td><code>double strtod( const char *nPtr, char **endPtr )</code></td>
<td>Converts the string nPtr to double.</td>
</tr>
<tr>
<td><code>char *strcat( const char *s1, const char *s2)</code></td>
<td>Appends string s2 to string s1.</td>
</tr>
<tr>
<td><code>char *strncat( const char *s1, const char *s2, size_t n )</code></td>
<td>Appends at most n characters of string s2 to string s1.</td>
</tr>
<tr>
<td><code>char *strcpy( const char *s1, const char *s2)</code></td>
<td>Copies string s2 to string s1.</td>
</tr>
<tr>
<td><code>char *strncpy( const char *s1, const char *s2, size_t n )</code></td>
<td>Copies at most n characters of string s2 to string s1.</td>
</tr>
<tr>
<td><code>char *strrev( char *s )</code></td>
<td>Reverses the string pointed to by s.</td>
</tr>
</tbody>
</table>

Standard Input/Output Library Functions

- Functions in `<stdio.h>`
  - Used to manipulate character and string data

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getchar( void )</code></td>
<td>Reads a character from the standard input and returns it as an integer.</td>
</tr>
<tr>
<td><code>char *gets( char *s )</code></td>
<td>Reads characters from the standard input into the array s until a newline or end-of-file character is encountered.</td>
</tr>
<tr>
<td><code>int putchar( int c )</code></td>
<td>Prints the character stored in c.</td>
</tr>
<tr>
<td><code>int puts( const char *s )</code></td>
<td>Prints the string s followed by a newline character.</td>
</tr>
<tr>
<td><code>int sprintf( char *s, const char *format, ...)</code></td>
<td>Equivalent to <code>printf</code>, except the output is stored in the array instead of printing it on the screen.</td>
</tr>
<tr>
<td><code>int sscanf( char *s, const char *format, ... )</code></td>
<td>Equivalent to <code>scanf</code>, except the input is read from the array instead of reading it from the keyboard.</td>
</tr>
</tbody>
</table>

A Non-Recursive Version of reverse()

In the previous example, the function reverse() was written recursively. Here is a non-recursive version of the function:

```c
void reverse( const char * const sPtr ){
    int i=0;
    /* print the string from end to beginning */
    for (; sPtr[i] != '\0'; i++) {
        if (sPtr[i] != '\n')
            putchar(sPtr[i]);
    }
}
```

String Manipulation Functions of the String Handling Library

- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Tokenize strings
  - Determine string length

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char *strncpy( char *s1, const char *s2, size_t n )</code></td>
<td>Copies at most n characters of string s2 into array s1. The value of s1 is returned.</td>
</tr>
<tr>
<td><code>char *strcat( char *s1, const char *s2)</code></td>
<td>Appends string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The value of s1 is returned.</td>
</tr>
<tr>
<td><code>char *strrev( char *s )</code></td>
<td>Reverses the string pointed to by s.</td>
</tr>
<tr>
<td><code>char *strnrev( char *s, const char *s2, size_t n )</code></td>
<td>Reverses at most n characters of string s2 into array s1. The value of s1 is returned.</td>
</tr>
</tbody>
</table>
Comparing Character Strings
Consider the following example:

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

int main()
{
    const char *string2 = "aehi lsTuv";
    const char *string1 = "The value is 3.14159";

    int status = strspn(string1, string2);
    printf("The length of the initial segment of string1 containing only characters from string2 = ", status);
    printf("The length of the initial segment of string1 = ", strspn(string1, string2));
}
```

Why?? Because the values of the pointers are being compared, not the contents of the strings.

Comparison Functions of the String Handling Library
- Comparing strings
  - Computer compares numeric ASCII codes of characters in string
  - Appendix D has a list of character codes

```c
int strcasecmp(const char *s1, const char *s2);

char *strchr(const char *s, int c);

size_t strlen(const char *s);

int strcoll(const char *s1, const char *s2);

char *strcat(char *str1, const char *str2);

char *strchr(const char *s, int c);

size_t strlen(const char *s);

int strncmp(const char *s1, const char *s2, size_t n);

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

size_t strlen(const char *s);

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

size_t strcspn(const char *s, const char *s2);

size_t strcspn(const char *s, int c);
```

Search Functions of the String Handling Library
- To determine if two character strings are equal, they must be compared character-by-character
- C has library functions to perform character string comparisons

```c
char *strcat(char *s1, const char *s2);

char *strchr(const char *s, int c);

int strcmp(char *s1, const char *s2, const char *s3);

int strcoll(const char *s1, const char *s2);

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

char *strchr(const char *s, int c);

size_t strlen(const char *s);

int strcoll(const char *s1, const char *s2);

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

Program Output
- The length of the initial segment of string1 containing only characters from string2 = 3
- The length of the initial segment of string1 = 3
Other Functions of the String Handling Library

- `char *strerror( int errornum );`
  - Creates a system-dependent error message based on `errornum`
  - Returns a pointer to the string
- `size_t strlen( const char *s );`
  - Returns the number of characters (before NULL) in string `s`

An Example

- Character and string processing example
- Use standard char/string library functions (fgets(), strlen(), strcpy, etc)
- Top-down design, with divide-and-conquer AND step-wise refinement
- Use a new “Unix” specific utility (“pipe”)

Problem Statement

- Take as input a string of characters (command line argument), and search for words as substrings
- Example:
  ```
  ./a.out thisisnotatest
  hi
  is
  is
  no
  at
  his
  not
  ate
  this
  test
  ```

Understanding the Problem and Proposing an Algorithm/Solution

Input string example: test
All possible (contiguous) sub-strings:
  t, e, s, t
te, es, st
tes, est
test

Refinement

- Take string input (argv[])
Refinement
For each "len"gth 1 to n  (n == length of input string)
  For each character i in input string:
    get each substring at position i, length "len"

Refinement
• For len = 1 to strlen(input)
  for i = 0 to strlen(input)
    get_chars(input,temp,i,len)

Refinement --
get_chars(source, destination, position, length);
• Function to pass in the source string, the
  destination string, the position in the source
  string, and the length of characters to acquire

get_chars() refinement
void get_chars(char *source, char *dest, int start, int length) {
    destination_start = 0;
    for i = start to length
    {
        dest[destination_start] = source[i];
        destination_start++
    }
    dest[destination_start] = '0';
}

Algorithm

#include<stdio.h>
void get_chars(char *source, char *dest, int start, int len);
Int main(int argc, char* argv[]) {
    char temp[30];
    int i = 0;
    int j = 0;
    int length = 0;
    for(length=1;length<=strlen(argv[1]);length++) {
        for(i=0;i<=strlen(argv[1])-length;i++)  {
            get_chars(argv[1],temp,i,length);
            printf("temp = %s
",temp);
        }
    }
}
Output

% more file
test
% cat file | ./testProgram.exe
word = test

How do we compare these substrings to a dictionary?

%cat file
● Directs the contents of “file” to the “standard input” -- the contents are printed to the screen
● This is a mechanism that we can use to send the contents of a large dictionary file to our program
● Unix pipe -- “|”
%cat file | ./testProgram.exe
● Instead of sending the contents of the file to the screen, we can “pipe” them to our program

Refinement

● Take string input (argv[0])
● Generate all substrings
● Compare all generated substrings to a “dictionary” to determine if substring is a “real” word

Becomes:
● Take string input (argv[0])
● Read all words of dictionary into an array
● Generate all substrings
  ▪ Compare each substring to the words in array
  ▪ If equal, then print ‘word’ that has been “found”

Simple program to show how a “pipe” may work

#include <stdio.h>

int main(int argc, char* argv[]) {
    char word[5];
    scanf("%s", word);
    printf("word = %s
", word);
}

Expand the program to handle a large number of words:

#define NUM_WORDS 45427
#define MAX 30  // max word length
#include<stdio.h>

int main(int argc, char* argv[]) {
    char word[5];
    scanf("%s", word);
    printf("word = %s
", word);
}
Even simpler method for input/output

- cat is most useful when directly taking the output from one program and sending it as the input to another
- If file already exists, use the Input Redirection Shell Command, <
  - `% wordFind.exe argvstuff < dictionary.txt`
- Can do the same for output (Output Redirection Shell Command)
  - `% wordFind.exe argvstuff < dictionary.txt > output.txt`

Now it is just a matter of combining the two partial solutions:

```c
/* read in a string from the command line 
and try all permutations against a dictionary */
#define NUM_WORDS 45427
#define MAX 30  // max word length
#include<stdio.h>

void get_chars(char *source, char *dest, int start, int len);

int main(int argc, char* argv[]){
  char words[NUM_WORDS][MAX];
  char temp[MAX];
  int i = 0, j=0, length=0;
  while(i<NUM_WORDS) {
    fgets(words[i],MAX,stdin);
    i++;
  }
  for(j=0;j<NUM_WORDS;j++) {
    for(i=0;i<MAX;i++) {
      if(words[j][i]=='\n'){ // remove newlines
        words[j][i]='\0';
      }
    }
  }
  for(length=1;length<=strlen(argv[1]);length++) {
    for(i=0;i<=strlen(argv[1])-length;i++) {
      get_chars(argv[1],temp,i,length);
      //printf("temp = %s\n",temp);
      for(j=0;j<NUM_WORDS;j++) {
        //printf("%s %s\n",temp,words[j]);
        if(strcmp(temp,words[j])==0){
          printf("%s\n",temp);
        }
      }
    }
  }
}

void get_chars(char *source, char *dest, int start, int len){
  int i = 0;
  int dstart = 0;
  for(i=start,dstart=0;dstart<len;i++,dstart++)   {
    dest[dstart] = source[i];
  }
  dest[dstart] = '\0'; // set string terminator
}
```

Future Modifications

- Modify to try all possible combinations of substrings:
  - the = t, h, e, te, he, ht, et, eh, the, teh, hte, het, eht
- Read from file (instead of pipe/stdio)
- Keep dictionary in memory and do multiple words