Characters and Strings

Characters

- Characters are the fundamental building blocks of source programs
  - Traditionally, a character variable is stored in a single byte of memory
  - Each character is encoded as a number using the 7-bit ASCII encoding
  - If you use a different locale, the character may take 2 bytes (think about Chinese characters); this has implications in how you allocate and use memory

- Character constants
  - One character surrounded by single quotes
  - ‘A’ or ‘?’
  - Since characters are encoded as integers, they allow arithmetic and logical operations; the following are legal:
    - `'a' + 2 // Evaluates to 'c'`
    - `'a' < 'c' // Evaluates to true`

- Characters are declared by the keyword char
  - Some operations on characters
    ```c
    char ch;
    for ( ch = 'a'; ch <= 'z'; ch++ )
      printf ( "%c", ch );
    ch = getchar(); // read in a character from stdin
    ```
  - Since a character is an 8-bit byte, it can hold any value from -128 to 127
  - You can use the keyword unsigned to keep only positive values
    - * unsigned char ch; will allow ch to hold any value in the range [0, 255]

- Special characters and non-graphic characters
  - Denoted by preceding other characters with a backslash \ 
    ```
    \n    newline
    \t horizontal tab
    \v vertical tab
    \b backspace
    \r carriage return
    \f form feed
    \\ backslash
    \' single quote
    \a alert
    ```
  - Another form \ddd where each d is an octal digit
    - * ddd specifies the desired internal value of a character
Strings

- NULL character
  * Indicated by the escape sequence \0
  * All bits corresponding to the character are zero
  * Not the same as the ASCII character 0 which is represented by hexadecimal sequence 30

- String constants
  - Also known as literals
  - Sequence of characters surrounded by double quotes
  - Backslash can be used for special characters
  - Double quotes within the string written as "
  - "ABC" or "%d\n\n"
  - A null character (\0) is added immediately after the final character of a string
    * "ABC" is stored in four bytes as "ABC\0"
    * String constant "A" is different from character constant ‘A’
  - String constant "ABC\nDEF" is a two-line string
  - **Important**: A character constant is enclosed in single quotes while string constants are enclosed in double quotes

**Fundamentals of strings and characters**

- There is no specific string type in C
  - Strings are arrays of char supported by library functions

- String is accessed via a pointer to its first character

- String is also viewed as an array of characters, with ‘\0’ being used to terminate the string

- A string variable is declared as a pointer to character (or array)

```c
char color[] = "blue";
char * color_ptr = color;
```

- What is the number of bytes reserved for the string in the above cases?

- You must declare enough space for the string (especially if you intend to increase the size of the string later on)

- Never forget to account for the NULL character when allocating space for strings

- Assigning a string to another variable
  - Since strings are implemented by a pointer to the first element of the character array, they cannot be copied by a simple assignment
  - `color_ptr = color` does not copy the string into `color_ptr` but merely copies the pointer value
  - You may have to use a function such as `strcpy` to actually achieve the copy operation
Assuming that color_ptr has enough memory allocated, the following function can also achieve the string copy

```c
void copy_string ( char * color_ptr, char * const color )
{
    char * in = color;
    char * out = color_ptr;
    while ( *out++ = *in++ );
}
```

- A string can be read by using `scanf` and the `%s` format specifier, but we will resort to using `fgets` to read strings from `stdio` and files, and `sscanf` to read them from within memory

- Be careful about mixing characters and strings, especially when passing them as parameters to functions

- Initializing strings
  - Can be done using either array or pointer notation
    - char color[] = "blue";
      * Compiler allocates the space and copies literal into that space
      * This string can be modified
    - char * color = "blue";
      * Compiler just creates a pointer to the string literal
      * This string cannot be modified

**Character handling library**

- Standard library in C to work with characters and strings
- You must include the header file `ctype.h` to use these functions
- In the following table, the type of variable `c` is an `int`, with its value restricted to that in an `unsigned char` (or that of the predefined constant `EOF`)
Strings

---

### Character classification macros or Predicates

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalpha(c)</td>
<td>c is a letter, [a–z] or [A–Z]</td>
</tr>
<tr>
<td>isupper(c)</td>
<td>c is an uppercase letter, [A–Z]</td>
</tr>
<tr>
<td>islower(c)</td>
<td>c is a lowercase letter, [a–z]</td>
</tr>
<tr>
<td>isdigit(c)</td>
<td>c is a digit [0–9]</td>
</tr>
<tr>
<td>isxdigit(c)</td>
<td>c is a hexadecimal digit, [0–9], [a–f], or [A–F]</td>
</tr>
<tr>
<td>isalnum(c)</td>
<td>c is an alphanumeric character, a letter or a digit</td>
</tr>
<tr>
<td>isgraph(c)</td>
<td>c is a visible graphic character</td>
</tr>
<tr>
<td>isprint(c)</td>
<td>c is a printing character</td>
</tr>
<tr>
<td>iscntrl(c)</td>
<td>c is a control character or \b</td>
</tr>
<tr>
<td>isascii(c)</td>
<td>c is an ASCII character, code less than \x0200</td>
</tr>
</tbody>
</table>

### Character conversion macros

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toascii(c)</td>
<td>Masks c with an appropriate value so that c is guaranteed to be in the ASCII range \x00 through \x7f</td>
</tr>
</tbody>
</table>

### Character conversion functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toupper(c)</td>
<td>Convert c to its uppercase equivalent</td>
</tr>
<tr>
<td>tolower(c)</td>
<td>Convert c to its lowercase equivalent</td>
</tr>
</tbody>
</table>

- Program to illustrate the character functions – char_fns.c

---

### String conversion functions

- Available in general utilities library (stdlib)
- Useful to convert a string of digits to integer or floating point values
- In the following table, str represents a string (array) and ptr represents a pointer to a character

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atof ( str )</td>
<td>Convert string to double precision number</td>
</tr>
<tr>
<td>strtod ( str, ptr )</td>
<td>Convert string to double precision number</td>
</tr>
<tr>
<td>atoi ( str )</td>
<td>Convert string to integer</td>
</tr>
<tr>
<td>atol ( str )</td>
<td>Convert string to integer</td>
</tr>
<tr>
<td>strtol ( str, ptr, base )</td>
<td>Convert string to integer</td>
</tr>
</tbody>
</table>

- A word about the strxto? functions
  - If the value of ptr is not (char **)NULL, a pointer to character terminating the scan is returned to the location pointed to by ptr
  - If no number can be formed, *ptr is set to str and 0.0 is returned
  - base is of type int and, if its value is between 0 and 36, is used as the base for conversion
    * 0x or 0X are ignored if base is 16

### Standard I/O library functions

- Require the file <stdio.h> to be included
• These functions are: `getchar`, `gets`, `putchar`, `puts`, `sprintf`, `sscanf`

• Writing strings using `printf` and `puts`
  – Use `%s` conversion in `printf` to write strings
  – If the NULL character is missing, `printf` continues printing until it finds a NULL somewhere in memory
  – Use the conversion `%.*s` to print a part of the string
    ```
    char str[] = "Hello world";
    printf ( "First five characters are: %.5s\n", str );
    ```
  – Elimination of . will print the string in full, if `p` is less than the string length
  – If string is smaller than `p` characters, it is right justified
  – String can be left justified by using `-`, as in `%-*s`
  – Program `str_print.c`

String manipulation functions

• Require the inclusion of the standard string library `<string.h>`

• These functions operate on null-terminated strings

• These functions do not check for overflow of any receiving strings

• In the following table, `s1` and `s2` represent pointers to character type (or strings)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcat ( s1, s2 )</code></td>
<td>Appends a copy of strings <code>s2</code> to the end of string <code>s1</code></td>
</tr>
<tr>
<td><code>strncat ( s1, s2, n )</code></td>
<td>Appends at most <code>n</code> characters from <code>s2</code> to <code>s1</code></td>
</tr>
<tr>
<td><code>strcpy ( s1, s2 )</code></td>
<td>Copies <code>s2</code> to <code>s1</code> until the null character</td>
</tr>
<tr>
<td><code>strncpy ( s1, s2, n )</code></td>
<td>Copies <code>s2</code> to <code>s1</code> until the null character, or <code>n</code> characters</td>
</tr>
<tr>
<td><code>strdup ( s )</code></td>
<td>Duplicates string and returns pointer to the new string</td>
</tr>
<tr>
<td><code>strlen ( s )</code></td>
<td>Number of characters in <code>s</code>, not including the NULL character</td>
</tr>
</tbody>
</table>

• `strcat`, `strncat`, `strcpy`, and `strncpy` return a pointer to the null-terminated string `s1`

• In `strcpy` and `strncpy`, if the length of target string `s1` is more than the source string `s2`, `s1` is padded with NULL characters

• `strdup` automatically allocates space for the new string

• `strdup` returns a NULL if it cannot allocate space for the duplicate string

String comparison functions

• These functions return an integer which is
  ```
  0   if the two strings are equal
  > 0 if first string is greater than second string (in alphabetic order)
  < 0 if first string is smaller than second string (in alphabetic order)
  ```
**Strings**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcmp (s1, s2)</code></td>
<td>Lexicographically compare strings s1 and s2</td>
</tr>
<tr>
<td><code>strncmp (s1, s2, n)</code></td>
<td>Lexicographically compare first n characters of strings s1 and s2</td>
</tr>
<tr>
<td><code>strcasemp (s1, s2)</code></td>
<td>Same as <code>strcmp</code> but ignore case differences</td>
</tr>
<tr>
<td><code>strncasemp (s1, s2, n)</code></td>
<td>Same as <code>strncmp</code> but ignore case differences</td>
</tr>
</tbody>
</table>

- The functions do not compare characters following the NULL character in the strings
- Collating sequences are different in ASCII and EBCDIC

**String search functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strchr (s, c)</code></td>
<td>Returns a pointer to the first occurrence of character c in string s</td>
</tr>
<tr>
<td><code>strrchr (s, c)</code></td>
<td>Returns a pointer to the last occurrence of character c in string s</td>
</tr>
<tr>
<td><code>strpbrk (s1, s2)</code></td>
<td>Return a pointer in s1 to the first occurrence of any character from s2</td>
</tr>
<tr>
<td><code>strspn (s1, s2)</code></td>
<td></td>
</tr>
<tr>
<td><code>strcspn (s1, s2)</code></td>
<td></td>
</tr>
<tr>
<td><code>strstr (s1, s2)</code></td>
<td></td>
</tr>
<tr>
<td><code>strtok (s1, s2)</code></td>
<td></td>
</tr>
</tbody>
</table>

- `strchr` and `strrchr` return a pointer to NULL if the character c does not appear in the string s

**Memory functions**

- Declared in `<memory.h>` file

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memcpy (s1, s2, n)</code></td>
<td></td>
</tr>
<tr>
<td><code>memccpy (s1, s2, c, n)</code></td>
<td></td>
</tr>
<tr>
<td><code>memchr (s, c, n)</code></td>
<td></td>
</tr>
<tr>
<td><code>memcmp (s1, s2, n)</code></td>
<td></td>
</tr>
<tr>
<td><code>memset (s, c, n)</code></td>
<td></td>
</tr>
</tbody>
</table>