Starting Out with C++: Early Objects 5th Edition

Chapter 12

More About Characters, Strings, and the **`string`** Class

Topics

Topics (continued)

12.1 C-Strings

- C-string: sequence of characters stored in adjacent memory locations and terminated by **`NULL`** character
- The C-string

  `"Hi there!"`

  would be stored in memory as shown:

`H`	`i`		`t`	`h`	`e`	`r`	`e`	`!`	`\0`

Representation of C-strings

- As a string literal

```
"Hi There!"
```

- As a pointer to **`char`**

```
char *p;
```

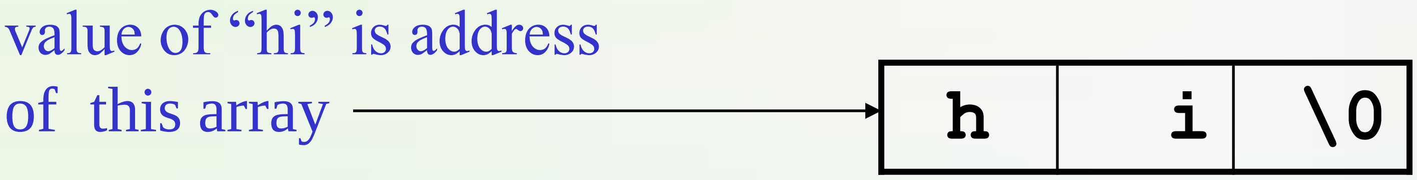
- As an array of characters

```
char str[20];
```

- All three representations are pointers to char

String Literals

- A string literal is stored as a null-terminated array of **`char`**
- Compiler uses the address of the array as the value of the string
- String literal is a pointer to char



Array of **char**

- Array of char can be defined and initialized to a C-string
```
char str1[20] = "hi";
```
- Array of char can be defined and later have a string copied into it
```
char str2[20];
strcpy(str2, "hi");
```

Array of **`char`**

- Name of array of char is used as a pointer to char
- Unlike string literal, a C-string defined as an array can be referred to in other parts of the program by using the array name

Pointer to **`char`**

- Defined as

```
char *pStr;
```

- Does not itself allocate memory
- Useful in repeatedly referring to C-strings defined as a string literal

```
pStr = "Hi there";
cout << pStr << " "
     << pStr;
```

Pointer to `char`

- Pointer to **`char`** can also refer to C-strings defined as arrays of char

```
char str[20] = "hi";
char *pStr = str;
cout << pStr;  // prints hi
```

- Make sure the pointer points to legitimate memory before using!

12.2 Library Functions for Working with C-Strings

- Require **`cstring`** header file
- Functions take one or more C-strings as arguments. Argument can be:
  - Name of an array of char
  - pointer to char
  - literal string

Library Functions for Working with C-Strings

- `int strlen(char *str)`

  Returns length of a C-string:

  ```
  cout << strlen("hello");
  ```

  Prints 5

`strcpy`

- `strcpy(char *dest, char *source)`

  Copies a string from a source address to a destination address

```
char name[15];
strcpy(name, "Deborah");
cout << name; // prints Deborah
```

`strcmp`

- `int strcmp(char *str1, char*str2)`

  Compares strings stored at two addresses to determine their relative alphabetic order:

  Returns a value:

  less than 0 if `str1` precedes `str2`

  equal to 0 if `str1` equals `str2`

  greater than 0 if `str1` succeeds `str2`

strcmp

- Often used to test for equality

```
 if(strcmp(str1, str2) == 0)
    cout << "equal";
else
    cout << "not equal";
```

- Also used to determine ordering of C-strings in sorting applications
- Note that C-strings cannot be compared using == (compares addresses of C-strings, not contents)

strstr

- `char *strstr(char *str1,char *str2)`

  Searches for the occurrence of `str2` within `str1`.

  Returns a pointer to the occurrence of `str2` within `str1` if found, and returns **`NULL`** otherwise

```
char s[15] = "Abracadabra";
char *found = strstr(s,"dab");
cout << found;      // prints dabra
```

12.3 String/Numeric Conversion Functions

- These functions convert between string and numeric forms of numbers
- Need to include the **`cstdlib`** header file

`atoi` and `atol`

- `atoi` converts **a**lphanumeric **to i**nt
- `atol` converts **a**lphanumeric **to l**ong

-
  ```
  int atoi(char *numericStr)
  long atol(char *numericStr)
  ```
- Examples:

```
int number; long lnumber;
number = atoi("57");
lnumber = atol("50000");
```

`atof`

- `atof` converts a numeric string to a floating point number, actually a double
- `double atof(char *numericStr)`
- Example:

```
double dnumber;
dnumber = atof("3.14159");
```

`atoi`, **`atol`**, **`atof`**

- if C-string being converted contains non-digits, results are undefined
  - function may return result of conversion up to first non-digit
  - function may return 0

`itoa`

- `itoa` converts an **int** **to** an **a**lphanumeric string
- Allows user to specify the base of conversion

```
itoa(int num, char *numStr, int base)
```

- `num` : number to convert
- `numStr`: array to hold resulting string
- `base`: base of conversion

itoa

```
itoa(int num, char *numStr, int base)
```

- Example: To convert the number 1200 to a hexadecimal string

  ```
  char numStr[10];
  itoa(1200, numStr, 16);
  ```

- The function performs no bounds-checking on the array `numStr`

12.4 Character Testing

- require **`cctype`** header file

FUNCTION	MEANING
`isalpha`	**`true`** if arg. is a letter, **`false`** otherwise
`isalnum`	**`true`** if arg. is a letter or digit, **`false`** otherwise
`isdigit`	**`true`** if arg. is a digit 0-9, **`false`** otherwise
`islower`	**`true`** if arg. is lowercase letter, **`false`** otherwise

Character Testing

- require **`cctype`** header file

FUNCTION	MEANING
`isprint`	**`true`** if arg. is a printable character, **`false`** otherwise
`ispunct`	**`true`** if arg. is a punctuation character, **`false`** otherwise

Character Testing

- require **`cctype`** header file

FUNCTION	MEANING
`isupper`	**`true`** if arg. is an uppercase letter, **`false`** otherwise
`isspace`	**`true`** if arg. is a whitespace character, **`false`** otherwise

12.5 Character Case Conversion

- require **`cctype`** header file
- Functions:
  - **`toupper`**: convert a letter to uppercase equivalent
  - **`tolower`**: convert a letter to lowercase equivalent

`toupper`

`toupper`: if `char` argument is lowercase letter, return uppercase equivalent; otherwise, return input unchanged

`toupper` actually takes an integer parameter and returns an integer result. The integers are the ascii codes of the characters

`toupper`

The function

```
    char upCase(int i)
    {return toupper(i);}
```

will work as follows:

```
char greeting[] = "Hello!";
cout << upCase[0]; //displays 'H'
cout << upCase[1]; //displays 'E'
cout << upCase[5]; //displays '!'
```

`tolower`

`tolower`: if `char` argument is uppercase letter, return lowercase equivalent; otherwise, return input unchanged

`tolower`

The function

```
char loCase(int i)
{return tolower(i);}
```

will work as follows

```
char greeting[] = "Hello!";
cout << loCase[0]; //displays 'h'
cout << loCase[1]; //displays 'e'
cout << loCase[5]; //displays '!'
```

12.6 Writing Your Own C-String Handling Functions

- When writing C-String Handling Functions:
  - can pass arrays or pointers to **`char`**
  - Can perform bounds checking to ensure enough space for results
  - Can anticipate unexpected user input

12.7 More About the C++ `string` Class

- The string class offers several advantages over C-style strings:
  - large body of member functions
  - overloaded operators to simplify expressions
- Need to include the **`string`** header file

`string` class constructors

- Default constructor `string()`
- Copy constructor `string(string&)` initializes string objects with values of other string objects
- Convert constructor `string(char *)` allows C-strings to be used wherever string class objects are expected
- Various other constructors

Overloaded `string` Operators

OPERATOR	MEANING
`>>`	reads whitespace-delimited strings into string object
`<<`	outputs string object to a stream
`=`	assigns string on right to string object on left
`+=`	appends string on right to end of contents of string on left

Overloaded `string` Operators (continued)

OPERATOR	MEANING
`+`	concatenates two strings
`[]`	references character in string using array notation
`>, >=, <, <=, ==, !=`	relational operators for string comparison. Return **`true`** or **`false`**

Overloaded `string` Operators

```
string word1, phrase;
string word2 = " Dog";
cin >> word1; // user enters "Hot"
              // word1 has "Hot"
phrase = word1 + word2; // phrase has
                        // "Hot Dog"
phrase += " on a bun";
for (int i = 0; i < 16; i++)
   cout << phrase[i];  // displays
                       // "Hot Dog on a bun"
```

`string` Member Functions

Categories:

- conversion to C-strings: `c_str`, `data`
- modification: `append`, `assign`, `clear`, `copy`, `erase`, `insert`, `replace`, `swap`
- space management: `capacity`, `empty`, `length`, `resize`, `size`
- substrings: `find`, `substr`
- comparison: `compare`

Conversion to C-strings

- `data()` and `c_str()` both return the C-string equivalent of a **`string`** object
- Useful in using a string object with a function that is expecting a C-string

```
char greeting[20] = "Have a ";
string str("nice day");
strcat(greeting, str.data());
```

Modification of **`string`** objects

- **`str.append(string s)`**

  appends contents of **`s`** to end of **`str`**
- Convert constructor for **`string`** allows a C-string to be passed in place of **`s`**

```
    string str("Have a ");
    str.append("nice day");
```

- **`append`** is overloaded for flexibility

Modification of `string` objects

- `str.insert(int pos, string s)`

  inserts `s` at position `pos` in `str`
- Convert constructor for `string` allows a C-string to be passed in place of `s`

  ```
  string str("Have a day");
  str.insert(7, "nice ");
  ```
- `insert` is overloaded for flexibility

12.8 Creating Your Own String Class

- A good way to put OOP skills into practice
- The class allocates dynamic memory, so has copy constructor, destructor, and overloaded assignment
- Overloads the stream insertion and extraction operators, and many other operators

Starting Out with C++: Early Objects 5th Edition

Chapter 12

More About Characters, Strings, and the `string` Class