

Chapter 8: Strings & Vectors

Instructor: Mark Edmonds

edmonds_mark@smc.edu

CStrings vs. `string`

- Below, we'll outline how you would directly store a string as an array of characters
- However, in general, it's much easier to use the `string` class to manipulate text data, as we have already been doing.
- This chapter is important for your understanding of C++ (and C), but I would recommend sticking to using the `string` class unless you have a good reason to use CStrings in C++

CStrings

- So far, we've dealt only with string literals such as "Hello, World!", but what if we want to store strings as variables?
- We'll use what's called a *C-style string* to do this

CStrings are arrays

- Just any array!
- We can write an array of characters to form a string:

```
1 char arr[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'};
```

- **But this is not a C-string**
 - This is an array of characters, but not a C-style string.
- Well what is a C-string?
 - A character array whose final character is the null character `\0`:
- To write "Hello World!" as a C-string:

```
1 char arr[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!', '\0'};
```

- But this is incredibly tedious to define strings this way
- Fortunately, we can assign a character array to string literal to create a C-string

```
1 char arr[] = "Hello, World!"; // arr will terminate with a null
    character.
2                               // Null character is automatically added
                               by the compiler
```

- Another example:

```
1 char t[5] = "HI";
```

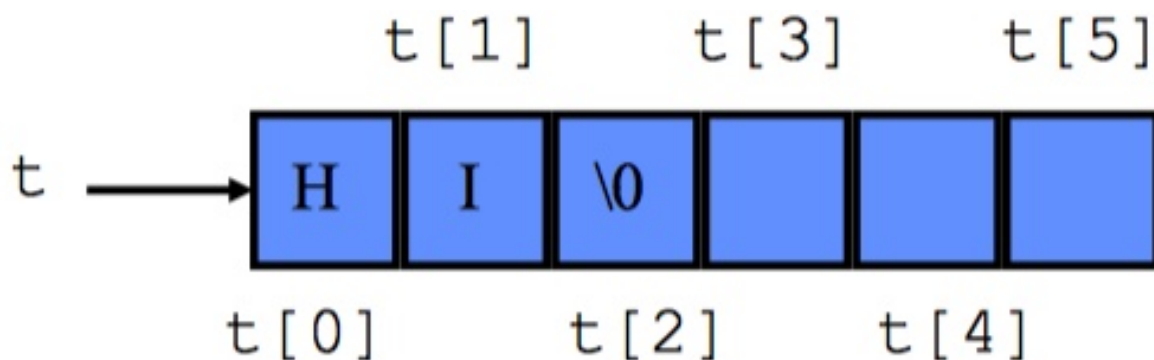


Figure 1: IMAGE

- When we initialize a character using a string literal, the null character is automatically added
 - This means the character array must have enough space for every character of the string plus an additional element for the null character.
 - * For instance, if we do the following, we don't end up with a C-string (there's no room for the entire string (and therefore there isn't room for the null character either)

```
1 char arr[5] = "Hello, World!"; // arr only stores the first 5 characters
2                               // arr has the values [H', 'e', 'l', 'l',
                               //                   ', 'o']
```

- But we also don't have to fill up the entire array either, the null-character indicates the end of the string.
- **Bottom line:** a character array is only a character array if it is **null-terminated**, meaning the final character is the null-character
- Why does any of this matter?
 - Strings are an incredibly common data type in real-world data.
 - Storing names, addresses, email addresses, etc all required strings.
 - There is a very large standard library header, called `<cstring>`, that provides a wide range of functionality for CStrings.

- * All of this functionality relies on using CStrings, not character arrays.

String Library

- Large library available for us to use to copy, compare, and manipulate strings.
- This is intended to help you, so you should view this as free functionality (as long as you are willing to read a tiny bit to figure out what the library functions do)
- Include the library with:

```
1 #include <cstring>
```

- The following table summarizes the CString library

Function	Meaning	Argument types	Return type
strcpy(dest, src)	dest = src	cstring, cstring	void
strcat(dest, src)	dest = dest + src	cstring, cstring	void
strlen(src)	length of cstring src	cstring	int
strcmp(s1, s2)	compares s1 and s2	cstring, cstring	int

Converting strings to other data types

- A bunch of functions to do this for you (included in `<cstdlib>`):
 - `atoi`: string to int
 - `atof`: string to float
 - `atol`: string to long
 - `strtod`: string to double
 - There are some more rare conversions provided by `<cstdlib>` as well

Example: CStrings.cpp

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 // Note the bizzarro way that CStrings are passed to functions
```

```
7 void dump( char song1[], char song2[], char song3[], char song4[], char
    song5[], char song6[] = "\n" );
8 void sort( char song1[], char song2[], char song3[], char song4[], char
    song5[], char song6[] = "\n" );
9
10
11 int main() {
12     // Let's play with strings
13     char ManilowTitle1[ 6 ] = "Mandy";
14     char ManilowTitle2[]    = "Could It Be Magic";
15     char ManilowTitle3[]    = { 'E', 'v', 'e', 'n', ' ', 'N', 'o', 'w',
        '\0' };
16
17     char ManilowTitle4[ 9 ];
18     strcpy( ManilowTitle4, "Daybreak" );
19
20     char ManilowTitle5[ 16 ];
21     strcpy( ManilowTitle5, "It's" );
22     strcat( ManilowTitle5, " A " );
23     strcat( ManilowTitle5, " Miracle" );
24
25     cout << "Here is your Barry Manilow songlist" << endl;
26     cout << "\t" << ManilowTitle1 << endl;
27     cout << "\t" << ManilowTitle2 << endl;
28     cout << "\t" << ManilowTitle3 << endl;
29     cout << "\t" << ManilowTitle4 << endl;
30     cout << "\t" << ManilowTitle5 << endl;
31
32     // We can do this with a function call
33     cout << "Here is your Barry Manilow songlist" << endl;
34     dump( ManilowTitle1, ManilowTitle2, ManilowTitle3, ManilowTitle4,
        ManilowTitle5 );
35
36     cout << "Here is your sorted songlist" << endl;
37     sort( ManilowTitle1, ManilowTitle2, ManilowTitle3, ManilowTitle4,
        ManilowTitle5 );
38
39     // Let's prompt for another song
40     char ManilowTitle6[80];
41     cout << "Here's your chance to add to the songlist!" << endl;
42     cin.getline( ManilowTitle6, 80 );
43
44     cout << "Here is your Barry Manilow songlist" << endl;
```

```
45     dump( ManilowTitle1, ManilowTitle2, ManilowTitle3, ManilowTitle4,
46           ManilowTitle5, ManilowTitle6 );
47
48     cout << "Here is your sorted songlist" << endl;
49     sort( ManilowTitle1, ManilowTitle2, ManilowTitle3, ManilowTitle4,
50           ManilowTitle5, ManilowTitle6 );
51
52     return( 0 );
53 }
54
55 void dump( char song1[], char song2[], char song3[], char song4[], char
56           song5[], char song6[] ) {
57     cout << "\t" << song1 << endl;
58     cout << "\t" << song2 << endl;
59     cout << "\t" << song3 << endl;
60     cout << "\t" << song4 << endl;
61     cout << "\t" << song5 << endl;
62     if (strcmp( song6, "\n" ) != 0)
63         cout << "\t" << song6 << endl;
64 }
65
66 void sort( char song1[], char song2[], char song3[], char song4[], char
67           song5[], char song6[] ) {
68     // This is very wasteful, but CS52 knows no other way
69     char songArray[6][30];
70     bool sentSix = false;
71     int total = 5;
72     strcpy( songArray[0], song1 );
73     strcpy( songArray[1], song2 );
74     strcpy( songArray[2], song3 );
75     strcpy( songArray[3], song4 );
76     strcpy( songArray[4], song5 );
77
78     if (strcmp( song6, "\n" ) != 0) {
79         strcpy( songArray[5], song6 );
80         sentSix = true;
81     }
82     if (sentSix)
83         total = 6;
84
85     // sort the array
86     for (int i = 0; i < total; i++) {
87         for (int j = i; j < total; j++) {
```

```

84         if (strcmp(songArray[j], songArray[i]) < 0) {
85             char temp[30];
86             strcpy( temp, songArray[j] );
87             strcpy( songArray[j], songArray[i] );
88             strcpy( songArray[i], temp );
89         }
90     }
91 }
92
93 for (int k = 0; k < total; k++)
94     cout << "\t" << songArray[k] << endl;
95 }

```

Stream input

- By default, when we use the stream insertion operator `>>`, it will eat whitespace (meaning any whitespace in the input won't be received by our programs)
- But whitespace can be meaningful in strings
- To read character data, we can use the `getline` function
- The `getline` function has the following signature

```

1 istream::getline(char s[], int i)

```

- This function reads up to `i-1` characters into `s` and will stop at a newline

```

1 const int LINESIZE=80;
2 char line1[LINESIZE];
3 char line2[LINESIZE];
4
5 cin.getline( line1, LINESIZE );
6 cin.getline( line2, LINESIZE );

```

Summarizing CStrings

- CStrings are not as nice to work with as `strings`
- At least you can always use loops to process character data, and look for the null-character `\0` to terminate the string

string class

- With C++, we have a much easier way to work with strings, as we have been
- The `string` class is provided by the `<string>` library
- The class allows for:
 - Concatentation using the `+` operator
 - Default and string argument constructor (can construct an empty string or initialize with a string literal)
 - Character access using the `[]` operator (indexing like an array)
 - `<<` and `>>` have been overloaded as you would expect (similar to how `cout` and `cin` operate)
 - All boolean operators work as you would expect

```
1 #include <string>
2
3 using namespace std;
4
5 int main() {
6     string name, dog("dog"), hotdog;
7     cin >> name;
8     hotdog = "hot " + dog;
9
10    for (int i=0; i < name.length(); ++i) {
11        cout << name[i] << " ";
12    }
13 }
```

getline for string objects

- `getline()` for `string` objects is a normal function, not a member of `istream`

```
1 string& getline(istream& input, string& str, char delimiter = '\n');
```

string member functions

Function	Meaning	Argument types	Return type
<code>substring(pos, len)</code>	substring starting at pos for length len	int, int	string
<code>empty()</code>	tests whether or not the string is empty	int, string	boolean

Function	Meaning	Argument types	Return type
<code>insert(pos, str)</code>	inserts str at pos	int, string	void
<code>remove(pos, len)</code>	remove starting at pos for length len	int, int	void
<code>find(str)</code>	find first occurrence of str in instance	string	int

Example: strings.cpp

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <cctype>
4  #include <string>
5
6  using namespace std;
7
8  void analyze( string s );
9  void peek( string s );
10
11 int main() {
12     // Let's play with string variables
13     // They are *so* much nicer than char *
14     string s;
15
16     cout << "Gimme a line of data to read" << endl;
17     getline( cin, s );
18     analyze( s );
19
20     return( 0 );
21 }
22
23 void analyze( string s ) {
24     int i = 0;
25
26     while (i < s.length() ) {
27         int locationOfSpace = s.find_first_of( " ", i + 1 );
28         if ( i == 0 )
29             peek( s.substr( i, locationOfSpace - i ) );
30         else
31             peek( s.substr( i + 1, locationOfSpace - i ) );
32         i = locationOfSpace;

```



```
33     }
34 }
35
36 void peek( string s ) {
37     bool isNumber = true;
38     bool isAlpha = true;
39     bool isUCase = true;
40     bool isLCase = true;
41     cout << s << "\t----> ";
42     for (int i = 1; i < s.length(); i++) {
43         if (isdigit( s.at( i ) ) || s.at(i) == '.' ) {
44             isNumber = (isNumber && true);
45             isAlpha = false;
46             isLCase = false;
47             isUCase = false;
48         }
49         else if (isalpha( s.at( i ) )) {
50             isNumber = false;
51             isAlpha = (isAlpha && true);
52             if (isupper( s.at(i) )) {
53                 isUCase = (isUCase && true);
54                 isLCase = false;
55             }
56             if (islower( s.at(i) )) {
57                 isLCase = (isLCase && true);
58                 isUCase = false;
59             }
60         }
61     }
62     if (isAlpha) {
63         cout << "looks ";
64         if (isLCase)
65             cout << "lowercase ";
66         else if (isUCase)
67             cout << "uppercase ";
68         else
69             cout << "mixed case ";
70         cout << "alphanumeric" << endl;
71     }
72     else if (isNumber) {
73         cout << "looks numeric" << endl;
74     }
75     else {
```

```
76         cout << " i can't make heads or tails of it!" << endl;
77     }
78 }
```

Vectors

- Vectors are similar to arrays, but their size can change size as your program runs.
- Much like how the `string` class is easier to use than CStrings, the `vector` class is much easier to use than native C++ arrays
- The `vector` class is included in the `<vector>`
- Vectors have a base type
- To declare a vector with the base type `int`, we would write:

```
1 vector<int> v; // creates a vector that can store ints
```

- Here, `<int>` identifies the template class
- You can use any base type in a template class

```
1 vector<string> v; // creates a vector that can store strings
```

- Similar to arrays and strings, vectors are indexed starting at 0, and we use `[]` to read or change values of a item:

```
1 v[i] = 42;
2 cout << v[i];
```

- But we can't use `[]` to initialize a new element (i.e. grow the vector)

Initializing vector elements

- We can use the member function `push_back()` to add an element to a vector:

```
1 vector<double> v;
2 v.push_back(0.0); // v contains [0.0]
3 v.push_back(1.1); // v contains [0.0, 1.1]
4 v.push_back(2.2); // v contains [0.0, 1.1, 2.2]
```

- We can also initialize multiple vector elements at a time:

```
1 vector<int> v(10); // allocates a vector with 10 default-initialized
    integers
```

- With this initialization, we can use `[]` to assign values to elements 0-9, and `push_back` will generate a new element in position 10

size of a vector

- With native arrays, we always had to keep track of how big our array was with a separate variable (e.g. `arr_len`)
- The `vector` class comes with a built-in member function `size` to return the number of elements in a vector
- To print every element of a vector, we might use:

```
1 for (unsigned int i = 0; i < v.size(); i++){  
2     cout << v[i] << endl;  
3 }
```

- We used an `unsigned int` instead of an `int` because:
 1. `unsigned ints` are nonnegative integers
 2. the `size()` function returns an `unsigned int`, so the compiler may issue a warning if the types don't match