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 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

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

- Another example:

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

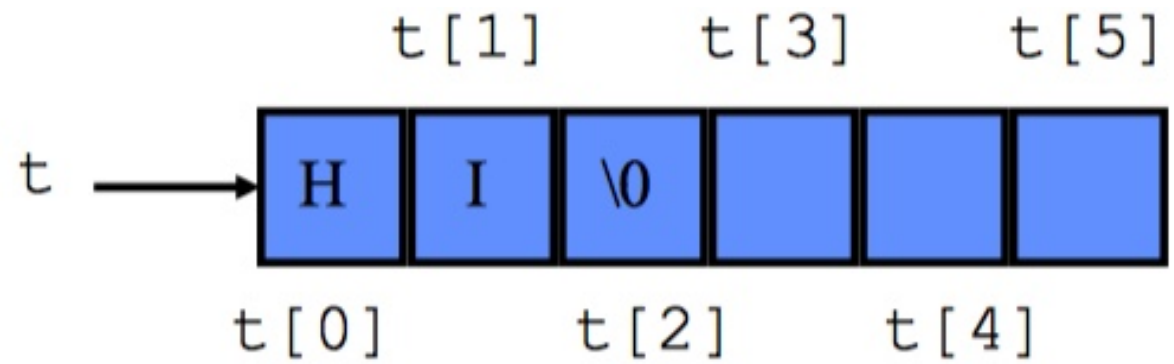

Figure 1: IMAGE

- 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)

```
char arr[5] = "Hello, World!"; // arr only stores the first 5 chracters
                               // 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:

```
#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`

```
#include <iostream>
#include <string>

using namespace std;

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

```
#include <string>

using namespace std;

int main() {
  string name, dog("dog"), hotdog;
  cin  >> name;
  hotdog = "hot " + dog;

  for (int i=0; i < name.length(); ++i) {
      cout << name[i] << " ";
  }
}
```

`getline` for `string` objects

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

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

`string` member functions

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

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

Example: `strings.cpp`

```
#include <iostream>
#include <cstdlib>
#include <cctype>
#include <string>

using namespace std;

void analyze( string s );
void peek( string s );

int main() {
    // Let's play with string variables
    // They are *so* much nicer than char *
    string s;

    cout << "Gimme a line of data to read" << endl;
    getline( cin, s );
    analyze( s );

    return( 0 );
}

void analyze( string s ) {
    int i = 0;

    while (i < s.length() ) {
        int locationOfSpace = s.find_first_of( " ", i + 1 );
        if ( i == 0 )
            peek( s.substr( i, locationOfSpace - i ) );
        else
            peek( s.substr( i + 1, locationOfSpace - i ) );
        i = locationOfSpace;
```

```
    }
}

void peek( string s ) {
    bool isNumber = true;
    bool isAlpha = true;
    bool isUCase = true;
    bool isLCase = true;
    cout << s << "\t----> ";
    for (int i = 1; i < s.length(); i++) {
        if (isdigit( s.at( i ) ) || s.at(i) == '.' ) {
            isNumber = (isNumber && true);
            isAlpha  = false;
            isLCase  = false;
            isUCase  = false;
        }
        else if (isalpha( s.at( i ) )) {
            isNumber = false;
            isAlpha  = (isAlpha && true);
            if (isupper( s.at(i) )) {
                isUCase = (isUCase && true);
                isLCase = false;
            }
            if (islower( s.at(i) )) {
                isLCase = (isLCase && true);
                isUCase = false;
            }
        }
    }
    if (isAlpha) {
        cout << "looks ";
        if (isLCase)
            cout << "lowercase ";
        else if (isUCase)
            cout << "uppercase ";
        else
            cout << "mixed case ";
        cout << "alphanumeric" << endl;
    }
    else if (isNumber) {
        cout << "looks numeric" << endl;
    }
    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 funciton `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:

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

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