Dynamic arrays / C Strings

- Dynamic arrays
  - syntax
  - why use
- C strings
  - Ex: command line arguments
- Call-by-pointer

Announcements

- Final exam: Tue, 5/11, 8am– 10am
  - closed book, closed note, etc.
  - alternate time TBA soon
- Review session Mon. 5/10 2 – 4 pm
- Review session info and finals week office hours will be posted on piazza soon
- Extra credit assignment available (due Tue 5/4)
- Course evals available through 5/4
  - via email from c-evals@usc.edu
  - or via blackboard.usc.edu

Class time for course evaluations

Your personalized link to online course evaluations:

- via email you received from c-evals@usc.edu
- or log into blackboard.usc.edu (look for Course Evaluations tab)

Additional Make material

- For the part we didn't get to in lecture last time:
  - do self-sudy: use make handout and lecture slides as a resource
  - answer to question on slide 26 from make lecture is in the file **`Makefile2`** (from **`04-27`** code on Vocareum)
  - makefile variables and default rules will not be on the final exam

Review: Fixed vs. dynamic arrays

- how to declare a fixed-size array of 100 int's?

- Declare a dynamic array of 100 ints (similar to Java arrays)

Why use dynamic arrays?

- Not for partially filled array applications:
  - Easier to use STL vector – do so when possible
- When you can't determine the array size at compile time
- Or if the array will need to change size during the run of the porgram.
- Lot’s of C code uses dynamic arrays:
  - C/C++ command-line arguments
  - C-string is a (possibly dynamic) array of chars

Arrays = pointers?

- In C/C++ pointers and arrays are almost interchangeable.
- An array is a pointer whose value you can't change.
- Example next slide.

Ex: Pointers vs. Arrays

```
char a[5];
char *b = new char[5];
a[0] = ‘x’;
b[0] = ‘y’;
cout << *a;
cout << *b;
cout << b[0];
a = b;  // illegal
b = a;  // legal
b[1] = ‘m’;
cout << a[1];    // m
a[3] = 'p';
cout <<  *(a+3); // same elmt. using pointer arith.
char *c = a+3;
```

Array parameters revisited

- Also, parameter types interchangeable. Can declare the following function either of these two ways:

```
void printArray1(int *arr, int size);
void printArray2(int arr[], int size);
```

- Can pass either static or dynamic array to the function:

```
int a[10];
int *b = new int[10];
printArray1(a, 10);
printArray1(b, 10);
printArray2(a, 10);
printArray2(b, 10);
```

Ambiguous pointer type?

- Suppose you see the declaration:

  ```
  Student * s;
  ```

- What data structure could `s` be?

C Strings

- Not an actual type in C
- C string means
  - an array of characters
  - but that is stored a particular way
- C library functions that operate on them assume that storage convention (<cstring>)
- Why do C++ programmers care?

C String representation

- A partially-filled array of characters such that,
- instead of keeping **`numChars`** as separate value, it uses a sentinel to mark the end of the string.
- more accurate: C string is a null-terminated array of chars
- The sentinel used is called the null char: **`'\0'`**

```
char str[10];
strcpy(str, "hello");          // copy a C string
```

- C string representation:

h	e	l	l	o	\0

- **`"hello"`** is a C string literal (internally has the null char)
- In the following, literal C string gets converted to C++ **`string`** object.

```
string s = "hello";
```

Dynamic C Strings

- Often C strings are dynamically allocated

```
char * str;

strcpy(str, “hello”);  // bad

str = new char[6];
strcpy(str, "hello");
int len = strlen(str);    // 5

char * str2 = new char[1];
strcpy(str2, "");  // empty string
// or   str2[0] = '\0';

str = str2;
```

C String library

- There are many C library functions for operating on C strings. Need **`#include <cstring>`**
- All of them have (more convenient) equivalents that work on C++ **`string`** objects
- Here are a few of them

C string function	"equiv." C++ **`string`** operation:
`strcpy` (doesn't alloc space)	**`=`** (*does* alloc space)
`strcmp`	**`==, <, >, etc.`**
`strlen`	**`length()`** (method)

- Avoid using C strings, switch to STL **`string`** if at all possible…

How to convert a C string to a `string`

- Scenario:
  I have code where someone passed me a C string . . .
- Use C-string to string constructor:

```
char *cstr;
. . .
string s(cstr);   // init string with C string
```

or

```
string t;
. . .
t = string(cstr); // put value in existing string
```

- Note: same constructor used to init with a literal string

```
string s("hello");
string s = "hello";
```

How to convert a `string` to a C string

- Scenario: I have to call a library function that only takes a C string (e.g., Linux library call) . . .
- There's a string method to convert to a C string:

```
string s;
// . . .
lib_func(s.c_str());
```

C String summary …

- While there are some functions (`<cstring>`) to operate on C-strings
- Unlike using `string` class…
  - Programmer is responsible for memory management (including space for null-char sentinel)
  - Programmer is often responsible for null-termination

Ex: command line arguments

- Arguments on command line are passed via parameters to **`main`**:

```
int main(int argc, char *argv[])
```

- **`argc`** number of arguments
- **`argv`** the arguments

Ex: command line arguments (cont.)

- Suppose we're writing the source code for compiler. Sample call:

```
g++ -ggdb –Wall foo.cpp
```

- In main for **g++** compiler:

```
int main(int argc, char *argv[])
```

- Values for this call:

```
argv[0]     “g++”
argv[1]     “-ggdb”
argv[2]     “-Wall”
argv[3]     “foo.cpp”
argc        4
```

Ex: command line arguments (cont.)

- You can make some assumptions:
- **`argc`** always **`>= 1`** (**`argv[0]`** is program name)
- **`strlen(argv[i])`** for valid **`i`** always **`>= 1`** (non-empty strings)
- What to do? convert to **`string`** right away
- Outline of code in **`main`** to process command line args:

```
for (int i = 1; i < argc; i++) {
  string arg(argv[i]);
  // process current arg
}
```

- Code to process the current arg can use **`string`** features like **`length()`**, **`substr()`**, **`=`**, **`==`**, **`[]`**, or **`+`**

Pass-by-pointer

- No call-by-reference in C (only C++)
- But can get the effect of call by reference
- Pass a pointer to the object / primitive variable
- Calling the function: use address-of (&)
- In the function: dereference the pointer (*)
- Example…

`swap` using pass-by-pointer

```
void swap(int * a, int * b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

Calling the function…

```
  int x = 10;
  int y = 20;
  swap(&x, &y);
```

`insertFront` using pass-by-pointer (instead of pass-by-reference)

```
void insertFront(ListType * listPtr, int newVal)
{
  Node * newGuy = new Node(newVal);
  newGuy->next = *listPtr;
  *listPtr = newGuy;
}
```

Calling the function…

```
ListType mylist = NULL;

insertFront(&myList, 3);
insertFront(&myList, 7);
```