COMP 1633: Intro to CS II

C-Strings

Charlotte Curtis

February 7, 2024

Where we left off

- Passing arrays to functions with and without `const`
- Partially filled arrays
- Sorting arrays
- Multidimensional arrays

> *Textbook Chapter 7*

```
int counts[N_LETTERS] = {};
char letter;
cin >> letter;

while (!cin.eof()) {
    if (is_alpha(letter))
        counts[to_index(letter)]++;
    cin >> letter;
}
```

Today's topics

- C-strings: a special kind of array
- C-string I/O
- C-string functions
- Separate compilation info

> *Textbook Section 8.1*

Multidimensional array passing

- Multidimensional arrays are **passed by reference** just like 1D arrays
- An initialization function might have the following **prototype**:

```
void initialize(char board[][COLS], int size);
```

- Like 1D arrays, the **first** dimension is **ignored**, however...
- The **second** dimension **must** be specified, and it **must** be a **constant**!

> *This is probably a good place to use a global constant*

Processing row by row

Depending on the data, you might want to process one row at a time:

```
const int MAX_RECORDS = 100;
const int NUM_FIELDS = 5;
int records[MAX_RECORDS][NUM_FIELDS] = {};

for (int row = 0; row < MAX_RECORDS; row++) {
    read_record(records[row], NUM_FIELDS);
}
```

- What should the prototype for `read_record` look like?
- How could you process **column by column**?

ND array check-in 1/2

The following function is intended to initialize a 2D array of integers to all -1. What is wrong with it?

A. Nothing, should work
B. `arr` is not passed by reference
C. A size is needed for the second dimension
D. The loop control variables are not initialized
E. `rows` and `cols` should be `const`

```
void initialize(int arr[][],
                int rows,
                int cols) {
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            arr[r][c] = -1;
        }
    }
}
```

ND array check-in 2/2

What is the output of the following code?

A. Nothing, compiler error

B. Nothing, runtime error

C. Random garbage

D. The memory address of `arr[][0]`

E. `0 0 0`

```
const int ROWS = 3;
const int COLS = 3;
int arr[ROWS][COLS] = {};

cout << arr[][0] << endl;
```

C-strings, finally!

- C-style strings are **arrays of characters**
- By now you know that this prints out the **memory address** of the array:

```
int primes[] = {2, 3, 5, 7, 11};
cout << primes << endl;
```

- But what about this?

```
char vowels[] = {'a', 'e', 'i', 'o', 'u'};
cout << vowels << endl;
```

- We've actually (almost) been using C-strings all along!

The null terminator

- Issue: how long should the string be?
- We *could* keep track of a partially filled array size, like this:

```
char vowels[5] = {'a', 'e', 'i', 'o', 'u'};
int size = 5;
```

- Or, we could use a **null terminator**:

```
char vowels[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
```

- The null terminator is a **sentinel** that marks the end of the string

> *An array of `char`s is not a C-string until it has a null terminator*

C-string shorthand

- C++ has a shorthand for initializing C-strings:

```
char vowels[] = "aeiou";
```

- The null terminator is **automatically** added
- The length is **one more** than the number of characters
- What happens in the following initializations?

```
char a_ch = 'a';
char a_str[] = "a";
char greeting[32] = "Hello!";
char hello[6] = "Hello!";
```

Some C-string gotchas

- Initializing with a string literal is a **shorthand** - the following are identical:

```
char message[] = "Hello!";
char message2[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

- This means that you **cannot** reassign a C-string, just as you can only use the curly bracket syntax when initializing an array
- You **can** reassign individual characters:

```
char message[] = "Hello!";
message[0] = 'G';
```

- Don't forget to allocate enough space for the null terminator!

C-string I/O

- Output is easy, we've been doing it all semester:

```
cout << "This is a C-string" << endl;
char message[] = "This is also a C-string";
cout << message << endl;
```

- Input is a bit more complicated:

```
char name[32]; // need to guess a size!
cout << "Enter your name: ";
cin >> name;
```

- Recall: what does `cin` do when it encounters whitespace?

The `getline` function

- All input streams (such as `cin`) have a `getline` member function

```
cin.getline(buffer, size, [delimiter]); // optional third argument
```

- This reads **up to** `size - 1` characters, **or** until the is encountered
- Default delimiter is the newline character
- The "buffer" is just a C-string that you provide

```
const int MAX_NAME = 32;
char name[MAX_NAME];
cin.getline(name, MAX_NAME);
```

> *If you enter more than than `size - 1` characters, they'll be left in the buffer!*

`get` vs. `getline`

- There's also `cin.get(buffer, size, delimiter)`
- They're almost the same, but `get` leaves the delimiter character in the buffer and `getline` consumes (and discards) it
- Both **do not ignore** leading whitespace (unlike `cin >> var`)
- If you need to skip over whitespace, there are a couple of options:
  - `cin.ignore(n)` to ignore the next `n` characters
  - `cin >> ws` to ignore all leading whitespace (my preference)

C-Strings plus functions

- We can pass C-strings to functions just like any other array
- Since a C-string always has a null terminator, we don't need to pass the size
- Example: write a function to calculate the length of a string

```
int len(const char str[]) {
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    return length;
}
```

- This is so common that C++ provides a function `strlen` in `<cstring>`

More `<cstring>` functions

- The `<cstring>` header provides useful functions for C-strings
- Some common ones are:
  - `strlen(str)`: returns the length of a C-string
  - `strcpy(dest, src)`: copies one C-string to another
  - `strcat(dest, src)`: concatenates two C-strings
  - `strcmp(str1, str2)`: compares two C-strings
- Caution: these functions **do not** check buffer size! For example, the following has **undefined behaviour** and will make your program behave strangely:

```
char name[4];
strcpy(name, "Charlotte Curtis");
```

Example: Hello World the complicated way

Python version

```python
hello = "Hello"
world = "World"

message = hello + " " + world + "!"
print(message)
```

C++ version

```cpp
char hello[] = "Hello";
char world[] = "World";

char message[32];
strcpy(message, hello);
strcat(message, " ");
strcat(message, world);
strcat(message, "!");

cout << message << endl;
```

`strcmp` behaviour

For the function call `strcmp(str1, str2)`, the return value is:

- `0` if `str1` and `str2` are equal (max length does not matter!)
- `-1` if `str1` comes before `str2` alphabetically
- `1` if `str1` comes after `str2` alphabetically

```
char fruit[];
cout << "What kind of fruit would you like? ";
cin >> fruit;

if (strcmp("apple", fruit) == 0) {
    cout << "Great choice, you can make pie!" << endl;
}
```

What about the `string` class?

- C++ provides a much more user-friendly `string` type
- You will encounter this in various tutorials, but for now, I want you to learn the pain of working with C-strings
- You will need C-strings and the `getline` function for Assignment 2
- **Do not use the `string` class for Assignment 2!**

Separate Compilation

- Typical lab structure:
  - `lab.h`
  - `lab.cpp` - `#include "lab.h"`
  - `main.cpp` - `#include "lab.h"`
- Prevents duplication of the code in `lab.h`, keeps main logic clear
- We could compile in multiple steps:
  - `g++ -c lab.cpp` - compiles `lab.cpp` into `lab.o`
  - `g++ -c main.cpp` - compiles `main.cpp` into `main.o`
  - `g++ -o main main.o lab.o` - links the two object files

What happens when you run `make`?

Compiling in multiple steps is annoying, so we dump it in a `makefile`

```
# This is "Makefile". Notice that comments begin with "#"
program: lab.o main.o
	g++ main.o lab.o –o program
main.o: main.cpp
	g++ -c main.cpp
lab.o: lab.cpp
	g++ -c lab.cpp
```

- Important: the indentation is a **tab**, not spaces! (emacs knows this)

Protecting against multiple `#include`s

- Most projects have many different modules (a somewhat random example)
- For example, in assignment 2 (not yet released):
  - `main.cpp` includes `applicant.h` and `score.h`
  - `score.h` includes `applicant.h`
- Problem: `#include` means "copy and paste" so we're defining stuff twice!
- Solution: **header guards**
  - Wrap your header file in `#ifndef` and `#endif` directives

Header guards

```
#ifndef APPLICANT_H
#define APPLICANT_H

... // contents of applicant.h

#endif // APPLICANT_H
```

- `#ifndef` checks if the macro `APPLICANT_H` is defined
- If it is, the preprocessor skips to the `#endif`
- By convention, the macro name is the header file name in all caps
- Also conventional to put a comment after the `#endif`

Separate Compilation check-in 1/2

Which of the following are good reasons to use separate compilation? Select all that apply.

A. It allows us to reuse code in multiple projects
B. It allows us to separate the main logic from other logical groupings
C. It prevents duplication of code
D. It prevents re-compiling code that hasn't changed
E. It allows us to use `make` to compile our code

Separate Compilation check-in 2/2

The `#include` directive is a preprocessor directive that means:

A. Check if a header has already been included, then include it
B. Copy and paste the contents of the header file into the source file
C. Cross-reference to the associated `.cpp` file
D. Compile the header file into an object file

Coming up next

- Lab: C-strings
- Next lecture: Structures
- Assignment 1 due TOMORROW!
- Assignment 2 available next week: Arrays, C-strings, and structures

> *Textbook Chapter 10*