CS 161
Introduction to CS I
Lecture 18

• Creating dynamic arrays
• Passing arrays to functions
Week 7 tips

• Study worksheet 7 is posted – give it a try after this lecture
• Assignment 4
  • Use valgrind to check for memory leaks (and other issues)
  • C-style strings: allocate enough room for the null character
    • strlen() does not include this character
  • Use the stack for local variables that will not grow/shrink. Use the heap for memory you need to pass around or change size over time.
Week 7 tips (2)

- Midterm 2 coming up on 2/28 – LINC 100
  - Covers material through end of week 7 (cumulative)
  - Practice questions will be posted by Monday 2/24 (week 8)
  - In-class review (but that's not all) on 2/26
  - Evening review session on 2/27, 6-7 p.m. in LINC 228
Static and dynamic memory

• Stack: memory is permanently allocated (within function) and permanently gone (when function exits)
  • "Gone" means that memory can be re-used (so no guarantee it will contain the original data)

• Heap:
  • Memory can be allocated when needed, freed when not needed
    • (e.g., each web page served; each document edited in a word processor)
  • Memory consumption can dynamically grow and shrink
    • Within a function
    • In different functions
A note about pointer arithmetic

• Increment a pointer in memory (e.g., to next item in an array):
  • p++;  
  • p += 2;
  • These statements **change** where the pointer is pointing

• Increment **the value** the pointer points to:
  • (*p)++;
  • (*p) += 2; /* () not required here, but a good idea */
  • These statements **do not change** where the pointer is pointing
Review static 1D arrays

1. const int n_people = 5;
2. int height[n_people];
3. for (int i=0; i<n_people; i++)
4.  height[i] = rand()%13 + 60;

• Note: allocating based on user input works too:
1. int n_people; cin >> n_people;
2. int height[n_people];
3. for (int i=0; i<n_people; i++)
4.  height[i] = rand()%13 + 60;

• But it cannot be changed later (different n_people)

See lec18-static-array.cpp
Review C-style strings

• C-style string: char array with '\0' (null) terminator
• Your turn: If the user types "Fred", what will this output?

```
1. char name[5] = {};  
2. cin.getline(name, 5); /* 5 includes '\0' */  
3. cout << name[0];  
4. for (int i=1; i<strlen(name); i++) {  
5.   cout << "_" << name[i];  
6. }  
7. cout << endl;
```
Why do we need a null terminator?

• The \0 (null) character indicates where the string ends in memory, just like the red bar on the grocery conveyer belt:
• If you omit it, many functions will not know when to stop
  • `strlen()`: when to stop counting?
  • `cout`: when to stop printing?
• You may get lucky if the memory after your array happens to be 0, but no guarantees
• `valgrind` will give an error for `strlen()`:
  • "Conditional jump or move depends on uninitialised value(s)"
C++ vs. C-Style strings

• What to #include
  • C++: `<string>`
  • C-style: `<cstring>` (C++ version of C's `<string.h>`)  

• Declaration
  • C++: `string`
  • C-style: `char[]`

• Access
  • `s.at(i)` or `s[i]`
  • `s[i]`

• Compatibility
  • C-style to C++: automatically converted
  • C++ to C-style: use `s.c_str()` to get a C-style string (char*) from `s`
Passing arrays to functions

• Arrays are always passed by reference (not value)
  • Why?
  • What does this mean for us?

1. int grades[5] = {90, 80, 85, 95, 100};
2. int max_grade = get_max(grades, 5); /* pass by ref */

• Assuming a function defined as one of the following:
1. int get_max(const int g[], const int n);
2. int get_max(const int* g, const int n);
Passing arrays to functions

1. int get_max(const int* g, const int n) {
2.  int m = g[0];
3.  for (int i=1; i<n; i++) {
4.    if (g[i] > m)
5.      m = g[i];
6.  }
7.  return m;
8.}

1. int main() {
2.  int grades[] = {90, 80, 85, 95, 100};
3.  cout << get_max(grades, 5) << endl;
4.  return 0;
5.}
Dynamic arrays (on the heap)

• Dynamic single item

1. float* f = new float;
2. . . .
3. delete f;
4. f = NULL;

• Dynamic array (e.g., when size could change)

1. float* g = new float[3]; /* from heap */
2. . . .
3. delete [] g; /* free the memory */
4. g = NULL;
Dynamic arrays

• Allow us to allocate and release memory as needed
• Web server: Instead of storing all possible web pages forever, only allocate space when it is served and release when that page is no longer in use
Stack and heap arrays

• Given these declarations:

1. int stack_arr[5];
2. int* heap_arr;

• Let's write code to:
  1. Allocate 5 integers from the heap for heap_arr
  2. For each array (stack_arr, heap_arr):
     a. Set the item at index 2 to 42
     b. Print the item at index 2
     c. Increment the item at index 2
     d. Print the address of the first item
  3. Free the memory associated with heap_arr

See lec18-arrays.cpp
What ideas and skills did we learn today?

• The importance of the null terminator for C-style strings
• How to pass arrays to functions
• Why it is useful to declare a function parameter "const"
• How to declare 1D arrays on the heap
• How delete 1D arrays on the heap
Week 7 begins!

- Attend lab (laptop required)
- Read Rao Lesson 7 (pp. 165-166)
  Rao Lesson 8 (pp. 189-198)
  Rao Lesson 4 (pp. 71-74)
  Rao Lesson 6 (pp. 145-146)
- Study session Thursday 2/20, 6-7 p.m. in LINC 268
- Assignment 4 Peer Review (due Wednesday, Feb. 19)

See you Wednesday!

- Bring: [Name of] object you could model as a 2D array