CS 161
Introduction to CS I
Lecture 14

• Manipulating data in memory
  • Pointers
  • And how they differ from references
Week 5 Tips

• Midterm #2 will be cumulative
  • Pick up Midterm #1 review your answers and the solutions
  • Pick up (bring ID) at KEC 1148 by 2/14, or after that from my office
• Variable shadowing: good to know about so you can read and trace through code, not recommended style
• Assignment 3 questions?
• Next Monday: guest instructor for lecture + no office hours for Dr. Wagstaff

2/7/2020
Passing arguments to functions

- `int v = 3;`
- `void fn(int w);`
- `void fn2(int& w);`

- **Pass by value**: make a **copy**
  - `fn(v);`
- **Pass by reference**: pass the **address** of the variable
  - `fn2(v); /* NOT fn2(&v); */`
Your turn: predict the output

1. void get_max(int a, int b, int& m) {
2.     m = (a < b) ? a : b; /* ternary/conditional operator */
3. }

4. int main() {
5.     int f = 17, g = 19, mx = -1;
6.     get_max(f, g, mx);
7.     cout << mx << endl;
8.     return 0;
9. }
Your turn: predict the output

```cpp
1. void get_max(int a, int b, int& m) {
2.     m = (a < b) ? a : b;  /* ternary/conditional operator */
3. }

4. int main() {
5.     int f = 17, g = 19, mx = -1;
6.     get_max(f, g, mx);
7.     cout << mx << endl;
8.     return 0;
9. }
```

17

(be sure you read the code!)
Why pass arguments by reference?

• Consumes less space: no need to make a copy
  • (As is done when passing by value)
  • This matters more when you work with large objects
• Slightly faster: no need to make a copy
• **** Since you can modify their values in the function, this is one way to get multiple results from one function
Pass multiple arguments by reference

1. void split_string(string s, int i,
2.    string& s1, string& s2) {
3.    s1 = s.substr(0, i);    /* get from index 0 to i-1 */
4.    s2 = s.substr(i);       /* get from index i to end */
5. }
6. int main() {
7.    string input, first_part, second_part;
8.    cin >> input;
9.    split_string(input, 3, first_part, second_part);
10.   cout << input << " : " << first_part << ", "
11.    << second_part << endl;
12.    return 0;
13.}
Pointers!

• Pointer = variable that stores a memory location (address)
• Examples:
  • char* cptr;
  • int* iptr;
• If not initialized, could point to invalid memory location
  • You could write over your own data by accident
  • You could also get a segmentation fault (what does this mean?)
• Good practice:
  • char* cptr = NULL;
  • int* iptr = NULL;
Review: Pass by reference

```c++
1. void compute_sum(int a, int b, int& s) {
2.     s = a + b;
3. }

4. int main() {
5.     int x = 2, y = 3, sum = 0;
6.     compute_sum(x, y, sum); /* no &sum in function call */
7.     cout << sum << endl;
8.     return 0;
9. }
```
Pass arguments as pointers

```c++
1. void compute_sum(int a, int b, int* s) {
2.     *s = a + b; /* note *s to dereference */
3. }
4. 
5. int main() {
6.     int x = 2, y = 3, sum = 0;
7.     compute_sum(x, y, &sum); /* note &sum in call */
8.     cout << sum << endl;
9. }
```
Pass arguments as pointers

1. void compute_sum(int a, int b, int* s) {
2.   *s = a + b; /* note *s to dereference */
3. }

4. int main() {
5.   int x = 2, y = 3, sum = 0;
6.   compute_sum(x, y, &sum); /* note &sum (NOT *sum) */
7.   cout << sum << endl;
8.   return 0;
9. }

2/7/2020
Pass arguments by reference (compare)

1. `void compute_sum(int a, int b, int& s) {`  
2. `s = a + b;`  
3. `}`

4. `int main() {`  
5. `int x = 2, y = 3, sum = 0;`  
6. `compute_sum(x, y, sum); /* no &sum in function call */`  
7. `cout << sum << endl;`  
8. `return 0;`  
9. `}`
Memory operators

- & and * can be used to specify data types
  - `int& z = n; /* declare a reference (alias) */`
  - `int* p; /* declare a pointer */`
- & and * can also be used as **operators** in expressions to perform actions
  - &: address-of
    - `p = &n;`
    - `&n = 5234; /* not allowed! (what would it mean?) */`
  - *: dereference (value-of): access the value at memory address
    - `cout << *p << endl; /* read */`
    - `*p = 27; /* write/change */`
"It Was A Dark and Stormy Pointer": A Play

- int* witch;
- witch = NULL;
- int cat = 7;
- int dog = 3;
- int mouse = 1;
- cat = dog + mouse;
- mouse *= 2;

- witch = &cat; /* address-of */
- *witch = 5;
- dog = *witch; /* dereference */
- witch = &mouse;
- *witch = cat;
References versus Pointers

- Do not confuse "reference" (a data type) with "pass by reference" (something that happens when you call a function)
- **Reference**: an alias to some variable (permanent)
  - `int& r = s;`
  - Can assign new values to `r` (which is `s`), but cannot make `r` be an alias to another variable later
  - Must be initialized when declared
- **Pointer**: stores the address of some variable
  - `int* p = &s;`
  - Can change what address `r` contains (where it points to) anytime
  - Can be declared, then initialized later
What vocabulary did we learn today?

• Pointers
• & (address-of) operator
• * (dereference) operator
What ideas and skills did we learn today?

• How to declare pointers
• How to pass pointers as function arguments
• How to trace through memory values when pointers are used
Week 5 nearly done

- Attend lab (laptop required)
- Read Rao Lesson 8 (pp. 177-186) – pointers and memory and https://www.geeksforgeeks.org/pointers-vs-references-cpp/
- Finish up Assignment 3 (due Sunday, Feb. 9)

Guest lecture on Monday!