CS 161 Lab #6 – Pointers and Dynamic Memory Allocation

- Start by getting checked off for (up to) 3 points of work from Lab 5 in the first 10 minutes.
- Your goal is to get all points for Lab 6 checked off by a TA by the end of this lab, so that you do not need to do additional work outside of the lab.
- You will work in small groups to complete the programming assignments, with each person writing their own program. Include each partner's name in your file header.

Goals:

- Refresh what references are and how/when to use them
- Practice using pointers
- Practice dynamic memory allocation and understanding the computer memory model

---

(2 pts) A. Reference Variables

Create a file called `lab6.cpp` with a `main()` function (and a file header with your name, the lab, and the date).

We have seen that passing a function argument by reference allows the function to change the original variable. Effectively, the function parameter becomes an **alias** for the argument that is passed in. We can also create aliases between variables without a function call, using & after the data type:

```
int raw_fish = 10;
int& sashimi = raw_fish;  /* sashimi: alias to raw_fish (same variable) */
```

Now changes to this variable can be made using either name (there are two names for the same variable).

1. In your `main()` function, declare `raw_fish` and `sashimi` as shown above.
2. Print the variable using each name:

```
cout << "raw_fish: " << raw_fish << endl;
cout << "sashimi: "  << sashimi << endl;
```

3. Set `sashimi` to -3 and print the variable using each name again.
4. Increment `raw_fish` by 100 and print the variable using each name again.
5. Are the results what you expect? Discuss them with your lab partners.

(2 pts) B. Passing Aliases with Pass-by-Reference

If you are working with other students, **add their names to your file header.**

1. Do you recall how to pass function arguments by reference? Above `main()`, define a function that takes in a string and counts how many periods it contains, using pass-by-reference so that `n_periods` (which is an alias) can be updated inside the function:

```
void count_punctuation(string s, int& n_periods) {
   n_periods = 0;   /* reset to start at 0; very important */
   /* write a for loop over string s to count periods */
```

```
    ...
}
```

Useful string functions:

- `s.length()` : the number of characters in string `s`
- `s[i]` : the character at position (index) `i` in string `s`

3. Identify **at least 3 test cases** and include them in the function header (with expected results).

4. Inside `main()`, declare 3 string variables to store your 3 test cases, then call your function on each string to count its periods. You will also need to declare an integer variable to store the number of periods before passing it to your function. In `main()`, print out how many periods each string contains, e.g. for a string `s1`:

```
cout << "<" << s1 << "> has " << np << " periods." << endl;
```

Do the results match your expectations? If not, debug the function until the output is correct.

5. Try out your test cases and ensure you get the correct results.

(3 pts) C. Passing Addresses with Pointers

1. Write a new function that does the same thing, this time using **pointers**:

```
void count_punctuation_ptr(string s, int* n_periods) {
    ...
}
```

What needs to be changed when you make the function call? What needs to change inside the function?

- Tip: To increment the value pointed to by `n_periods`, use parentheses to ensure that the dereference happens before the increment:
- `(*n_periods)++;  /* normally, ++ has higher precedence than *   */`

2. Run it with the same test cases you created in Part B.

3. Since this function can be written with either references or pointers, what are the advantages/disadvantages to each approach? Which one (at this time) makes more intuitive sense to you?

(3 pts) D. Working with Dynamic Memory

So far we have used **static memory allocation**, where memory for variables comes from the stack. In this part you will practice allocating and freeing **dynamic memory** from the heap, which has to be explicitly freed when you are done using it.

1. Inside your `main()` function, **allocate 3 variables from the heap**, each of a different type, using `new`. Here is an example to get you started:

```
  short* cookie = new short; /* allocate a new short from the heap */
                             /* cookie is a pointer to a short */
```

2. Assign values to each of your 3 variables. We have to dereference each pointer with the * operator to assign a value to the variable. If we assigned to `cookie` instead of `*cookie`, that would change the address stored in `*cookie`. Instead, do this:

```
  *cookie = 44;  /* use *cookie to dereference cookie and assign 44 */
```

3. Compile your program and run valgrind (at the command line) on your program and observe what it says about memory leaks.

```
$ g++ lab6.cpp -o lab6
$ valgrind lab6
```

In this example, I allocated a `short` (2 bytes), an `int` (4 bytes), and a `long` (8 bytes), for a total of 14 bytes, all of which appear as memory leaks:

```
==22571== HEAP SUMMARY:
==22571==     in use at exit: 14 bytes in 3 blocks
==22571==   total heap usage: 3 allocs, 0 frees, 14 bytes allocated
==22571==
==22571== LEAK SUMMARY:
==22571==    definitely lost: 14 bytes in 3 blocks
```

4. Fix the memory leaks by freeing the heap memory you allocated, using `delete`, e.g.:

```
  delete cookie; /* free heap memory used to store cookie */
```

5. Run valgrind again and confirm that the memory leaks are gone (the "`LEAK SUMMARY`" should be replaced with "`All heap blocks were freed -- no leaks are possible`"). If you still see memory leaks, check to make sure you used `delete` on each of your 3 variables.

6. Run your program using PythonTutor: http://pythontutor.com/visualize.html
- Select "C++" from the "Write code in" dropdown menu
- Paste your program (including function definitions and main()) into the box
- Click "Visualize Execution"
- Click "next" to move forward and examine what we see in memory ("stack").
- You can click on a particular line of code to jump to that point in the visualization.

E. Get your work checked off by a TA.

Submit your program on Canvas (can be done while you are waiting to be checked off). All partners' names must be in the comment header to get credit.

1. Transfer your .cpp file from the ENGR servers to your local laptop.
2. Go to the Lab 6 assignment: https://canvas.oregonstate.edu/courses/1770357/assignments/7847928
3. Click "Submit Assignment".
4. Upload your .cpp file (lab6.cpp).
5. Click the **"Submit Assignment"** button.
6. You are done!

If you finish the lab early, this is a golden chance to work on your Assignment 4 design (with TAs nearby to answer questions!).

Remember, the assignment you submit must be your work alone (no partners).