CSCI 2270
Data Structures and Algorithms
Lecture 4

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 112/128
Wed 1-2pm
Thurs 2-3pm

Getting the most out of the CS department and the school

- Undergraduate groups in computer science
  - http://innovatecu.com/ Undergraduate events
  - http://wic.cs.colorado.edu/ Women in Computing
  - CU ACM student chapter (link TBD)
  - Game Development club (link TBD)
- Departmental talks (colloquia)
  - http://www.colorado.edu/cs/colloquia/colloquium-schedule
  - A few of these involve our own faculty
- Other departmental talks
  - IQBio: http://biofrontiers.colorado.edu/events/upcoming-events
  - Linguistics: http://www.colorado.edu/linguistics/talks/
  - Electrical Eng: http://ecee.colorado.edu/news/seminars.html

++n vs n++

++n is prefix increment

    increment n and then use the increased value

n++ is postfix increment

    use the original value and then increment n

When n++ or ++n are single statements, same effect

```
n++;
++n;
```

It turns out that ++n can be fractionally faster for certain data types. (Doesn't need to copy original.)

++n vs n++, example 1

```
void plus_plus_example_1()
{
	int n = 5;
	while (n < 20)
	{
		cout << ++n << " ";
		cout << n++ << endl;
	}
}
```

++n vs n++, example 2

```
void plus_plus_example_2()
{
	int b[4];
	b[0] = 1; b[1] = 3; b[2] = 9; b[3] = 27;

	int n = 0;
	while (n < 4)
		cout << b[n++] << endl;
	n = 0;
	while (n < 4)
		cout << b[++n] << endl;
}
```

++n vs n++, example 3

```
void plus_plus_example_3()
{
      int b[4];
      int* b_ptr = &(b[0]);
      b[0] = 1;   b[1] = 2;   b[2] = 4;   b[3] = 8;
      for (int i = 0; i < 4; i++)
      {
            cout << *(b_ptr++) << endl;
      }
}
```

Recall the short but happy life of an integer at compile time

```
void example_with_a()
{
      int a = 5;            // a is born!
}                           // a is killed. :(
```

At the end of our function, just at the closing bracket }, that int a gets destroyed.

Destroying it means that *we no longer keep that memory reserved* on the stack for a. Now other variables can be written into a's former slots in the memory stack. We can't predict exactly when this will happen, but it's only a matter of time before something else writes data over this slot.

Integers at compile time

```
void example_loop_ok()
{
        for (int a = 5; a < 20; a++)
        {
                cout << a << endl;
        }
}
```

In this example, a only exists inside the for loop; at the end of the for loop, just at the closing bracket }, that int a gets destroyed.

Integers at compile time

```
void example_loop_not_ok()
{
      for (int a = 5; a < 20; a++)
      {
            cout << a << endl;
      }
      cout << a << endl;
}
```

What error will this produce?

Input variables, by value

```
int triple_a_number_1(int starting_number)
{
     starting_number *= 3;
     return starting_number;
}
```

The variable starting_number is passed in to the function; the compiler makes room for this integer in memory, just like it did for a in the previous slide.

What happens to this memory for starting_number at the }?

Input variables, by value

```
int triple_a_number_1(int starting_number)
{
     starting_number *= 3;
     return starting_number;
}

int main()
{
     int lebowski = 9;
     cout << triple_a_number_1(lebowski) << endl;
     cout << lebowski;
}
```

Input variables, by reference

```
int triple_a_number_2(int& starting_number)
{
    starting_number *= 3;
    return starting_number;
}

int main()
{
    int bob = 9;
    cout << triple_a_number_2(bob) << endl;
    cout << bob;
}
```

Input variables, by reference

```
int triple_a_number_2(int& starting_number)
 {
      starting_number *= 3;
      return starting_number;
 }
```

Here, we’re passing in the *address* of the starting number.

That means we have access to the original variable—any changes we make in this function affect variables like bob outside the function.

```
      int bob = 9;
      cout << triple_a_number_2(bob) << endl;
```

It also means that we don’t destroy anything at the end, unlike the previous example.

Input variables, by reference

```
int triple_a_number_2(int& starting_number)
{
   starting_number *= 3;
   return starting_number;
}
```

We know that starting_number's passed in by address, which means it's sort of like a pointer. (But it's not exactly like a pointer.)

So why don't we have to dereference this thing?

```
   *starting_number *= 3;          // no
   return *starting_number;        // also no
```

(In some environments, it's done automatically. Dissassembling the code would show you this. But please don't dereference any reference inputs yourself.)

Input variables, by const reference

```
int triple_a_number_3(const int& starting_number)
{
   starting_number *= 3;   // NO, compile error
   return starting_number;
}
```

Protects inputs from changing, while passing them by reference

Why do this? It’s not a great idea for passing an int. But if you have a big variable that takes up a lot of room, this lets you avoid copying it like a value parameter, while still protecting the original …

Normal return by value

```
int get_a_number_1()
{
  int answer;
  cout << "Tell me a number: " << endl;
  cin >> answer;
  return answer;
}
```

Normal return by value

```
int get_a_number_1()
{
  int answer;
  cout << "Tell me a number: " << endl;
  cin >> answer;
  return answer;
}
int main()
{
  int number = get_a_number_1();
  cout << "You entered  " << number << endl;
}
```

Dangerous return by reference

```
int& get_a_number_2()
{
  int answer;
  cout << "Tell me a number: " << endl;
  cin >> answer;
  return answer;
}
int main()
{
  int other_number = get_a_number_2();
  cout << "You entered " << other_number <<
endl;
}
```