

CS 246 Winter 2014 - Tutorial 4

October 7, 2014

1 Summary

- Pointers and References
- Memory Management
- Overloading

2 Pointers and References

- Let's review some basics about pointers and references:

```
int x = 42;
int *y = &x;
int &z = x;
// What do the following evaluate to?
x == z
x == y
z == y
&x == y
&x == &z
```

- Now let's add `const` into the mix. Which of the following variables can we change and which can change `x`?

```
const int x = 42;
const int *a = &x;
int * const b = &x;
const int * const c = &x;
```

- Now let's examine why we might want to use references over pointers:

```
int x = 4;
int y = 2;
int *p1 = &x, *p2 = &y;
int &r1 = x, &r2 = y;

*p2 = ((*p1 + *p2) * (*p2 - *p1)) / (*p2 - *p1);
// vs
r2 = ((r1 + r2) * (r2 - r1)) / (r2 - r1);
```

- Because we don't need to dereference references then we are less likely to make mistakes when we are writing code.

2.1 Pass-by-Value and Pass-by-Reference

- We say that an argument (to a function) is passed by value when it is copied
 - We may also call this passing by pointer (if the parameter is a pointer)
- We say that an argument (to a function) is passed by reference when it is a reference parameter
- Consider:

```
int foo(int *p, int q, int &r);
```

- Both p and q are passed by value and r is passed by reference
- However, p and r can both pass back changes
- Pass-by-const-ref occurs when we pass an argument as constant reference
- By doing so, we get 2 main benefits:
 - Large structures are not copied and can't be changed
 - Can pass in literal values

```
int foo(int &x, const int& y){...}
int main(){
    int a = 42;
    foo(a,a);
    foo(a,43);
    foo(43,a); // Invalid, what does it mean to change a literal?
    foo(43,43); // As above
}
```

3 Memory Management

- By default (in C++), memory is allocated on the stack


```
int x = 6;
// A single integer is allocated on the stack
int arr[10];
// 10 integers are allocated on the stack
```
- However, stack allocated memory is invalidated when you leave the scope of the block the memory was allocated in

```
int* foo(){
    int x = 42;
    if (x == 42){
        int y = 84;
    } // y is invalidated
    return &x; // x be invalidated
}
```

- To have memory persist between different scopes (e.g. between functions) then we need to allocate it on the heap
- In C++, operator **new** accomplishes this by taking in a type and allocating the appropriate amount of memory for the given type

```
int * x = new int;
int * y = new int [10];
delete x;
delete [] y;
```

- Heap allocated memory is freed using **delete** or **delete []**
- What happens if we delete an array using **delete**?
- **Warning: Never mix C dynamic memory management and C++ dynamic memory management. Bad things can happen.**
- Let's consider the example of taking an array (of size 10) and reversing it:
- First, let's look at a bad solution:

```
int * reverseCopy(int arr[]){
    int revArr[10];
    for(int i=0; i < 10; ++i){
        revArr[i] = arr[9-i];
    }
    return revArr;
}
```

- This obviously will not always behave the way we expect because the memory we allocated for `revArr` could be overwritten.
- Let's fix this with heap allocation

```
int * reverseCopy(int arr[]){
    int* revArr = new int[10];
    for(int i=0; i < 10; ++i){
        revArr[i] = arr[9-i];
    }
    return revArr;
}
```

- Now the reversed array will persist beyond the scope of the function.

4 Overloading

- **Overloading** occurs when a name has multiple meanings in the same context
- For routines, the *number* and *type* are used to select from a name's multiple meanings
- Note that in C++, **return type is never used used for overloading**
- Which of the following overloads are valid?

```
int foo(int x, int y); // Original
int foo(int x); // Valid
double foo(int x, int y); // Invalid
int foo(double x, int y); // Valid
int foo(double x, double y); // Valid
int foo (int x, int y, int z = 7); // Invalid
```

- Why is the last overload invalid?

5 Tip of the Week

- This week's tip is learning how to use the `screen` command in the Linux Environment
- Occasionally, you may find it useful to have multiple terminal windows open but for whatever reason, you decide you don't actually want two terminals open.
- There is a program called `screen` which is run using *screen*. Once you have `screen` running and you are back to the command line, there are a few options you can use.
 - `ctrl-a c`: Creates a new screen
 - `ctrl-a n`: Changes to the next screen
 - `ctrl-a ?`: Bring up a list of screen commands
- `screen` can be used to go between two terminals quickly without having to close the program being used in either terminal