CS 246 Fall 2015 - Tutorial 5

October 16, 2015

1 Summary

- Pointers and References
- Memory Management
- Overloading
- Preprocessor - #include guards

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?

```
int x = 42;
const int y = 42;
const int *a = &x;
int * const b = &x;
const int * const c = &x;
int const * d = &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 paremeter
- 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
  const int N = ...;
  int arr[N];
  // N 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 will 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 [N];
  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.**
- We should only use the heap when we absolutely have to:
  - The value created must outlive the scope of the variable storing it.
  - The size of a collection/array is unknown or likely to change.
  - The memory required is large (why?).
- Let's consider the example of taking an array (of size 10) and reversing it:
- First, let's look at a bad solution:

```
const int SIZE = 10;
int * reverseCopy(int arr[]){
  int revArr[SIZE];
  for(int i=0; i < SIZE; ++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

```
const int SIZE = 10;
int * reverseCopy(int arr[]){
  int* revArr = new int[SIZE];
  for(int i=0; i < SIZE; ++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 to determine a function overload**
- 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?

4.1 Operator Overloading

- Recall, that operators are actually functions and so can be overloaded
- For example,
  - `2+3` is actually `operator+(2,3)`
  - `string str1=...,str2=...;str1 + str2` is actually `operator+(str1,str2)`
  - `cout << "Hello"` is actually `operator<<(cout, "Hello")`
- This implies that we can define operators for user-defined structures

```
#include <iostream>
#include <string>
using namespace std;

const int maxStudents = 100;
struct ClassGrades{
  string names[maxStudents];
  double grades[maxStudents];
  string className;
  int numStudents;
};

ostream& operator<<(ostream& out, const ClassGrades& cg){
    out << cg.className << ": " << cg.numStudents << " students" << endl;
    for(int i=0; i < cg.numStudents; ++i){
      out << left << setw(31) << cg.names[i] << setw(5) << cg.grades[i] << endl;
    }

  // Always remember to return ostream. Why?
  return out;
}
```

5 Preprocessor

- Recall, that a preprocessor statement is any line that begins with #
- Also recall, Preprocessor statements are evaluated before the code makes it to the compiler
- We can have statements for file inclusion, substitution, and conditional inclusion
- Today, we're just going to focus on #include guards as they will become very important as the course goes on
- Two main goals of #include guards:
  1. Prevent the same code from being included multiple times
  2. Prevent cyclic includes (try to compile `cycle.cpp`)
- Accordingly, any header (.h) file you write should look like:

```
#ifndef SOMEHEADER_H
#define SOMEHEADER_H
  ... // function/data/class declaractions
#endif
```

- We'll see some more #include guards in the context of classes.