CS 246 Spring 2019 - Tutorial 4

June 5, 2019

1 Summary

- References
- Dynamic Memory
- Preprocessor
- Include Guards

2 References

- Syntax:

```
int x = 42;
int &rx = x;
```

- A reference is basically a dereferenced constant pointer to an object. What does this mean?
  - **Constant pointer to an object:** the object which a reference is refering to cannot be changed after initialization.
  - **Dereferenced:** When working with pointers, the pointer must be dereferenced to access the value of the object. For example, we can define `int *xp = &x;`. Then, to access the value pointed by `xp`, you must use `*xp`.
    References cannot be dereferenced (unless they're a reference to a pointer).
  - Consider the code below:

```
int x = 10, y = 5;

int &rx = x;
int &ry = y;

int *px = &x;
int *py = &y;

int res1 = (*px + *py) * (*px - *py);
int res2 = (rx + ry) * (rx - ry);
```

The two variables `res1` and `res2` contain the same value but the calculation with references looks simpler.

- A reference is an alias to an object. The reference and the object share the address which they are refering to in memory.

```
int x = 17;
int &rx = x;

cout << &x << endl;   // these two lines print the same address
cout << &rx << endl;
```

- Note: references cannot be null.

2.1 Pass-by-Reference

- **Pass-by-value:** makes a copy of the parameter passed for use during the function. Changes to the parameter do not exist outside of the scope of the function.
- **Pass-by-reference:** creates an alias to the object which is a parameter.
- Writing a function which take a pointer to a variable simulates pass-by-reference. (A copy of the address is made but changes to the variable persist after the function call.)
- Passing-by-reference is usually faster than passing-by-value because copying the parameter takes more time than creating a reference.
- Literals cannot be passed by reference since they are not *lvalues* (something that has an address), with the exception of pass-by-const-reference.
- **Pass-by-const-reference** occurs when we pass an argument as a constant reference. You can pass a literal as a const reference since there is no danger of the function changing the literal.
- By doing so, we get 2 main benefits:
  - Large structures are not copied and can't be changed
  - Can pass in literal values
- Example of pass-by-reference vs. pass-by-const-reference:

```
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 Dynamic Memory

- In C++, we use `new` instead of `malloc`, and `delete` instead of `free`. `new` allocates enough space on the heap for one object and returns a pointer to the allocated memory. `delete` frees the memory stored at the address the variable holds.

```
int *x = new int{5};
...
delete x;
```

- To allocate an array on the heap: `int *arr =  new int[10]`. To delete an array on the heap: `delete [] arr`.
- Note: make sure you always use `new` and `delete` together, and use `new[]` and `delete []` together.

4 Operator Overloading

- We can overload many of the built in operators including the arithmetic operators.
- This is often useful when implementing our own structures.
- When overloading operators, we should give our operators logical meanings. i.e. `operator+` should do something that resembles addition

4.1 Cascading

- We've gotten used to seeing code that looks like the following:

```
int num = 5;
cout << "The magic number is " << num * num - num << endl;
```

- We see that `num * num - num` execute `num * num` and the result will have `num` subtracted from it. What is happening here is that `operator*` is being called which returns an int, then `operator-` is being called and returns the int which is printed.
- Furthermore, `operator<<` returns the ostream it was given which is used later in the expression.
- When we write operators, we want our operators to behave in a similar fashion. This means that our operators will return the result of the operation.
- When writing input or output operators, an istream or ostream will be returned. This will always be a reference to the stream which was input as we do not want to copy streams.
- **Exercise:** Let's write `operator+`, `operator*`, `operator<<` and `operator>>` for the Rational structure.

5 Preprocessor

- The preprocessor runs before the compiler. It handles all preprocessor directives (any line which begins with `#`).
- Common preprocessor directives:
  - `#include "file.h"`: pastes the contents of `file.h`
  - `#define var`: defines a preprocessor macro variable.
  - `#ifdef var`: includes code if `var` is defined. Must be closed with an `#endif`
  - `#ifndef var`: includes code if `var` is not defined.

5.1 Include Guards

- As you learnt in CS 136, we often want to program in modules which logically separate our code. When we do this, we will often end up including header files in other files so that we have access to functions declared in a module.
- However, we may want to include the same file multiple times in our program which will result in compilation errors.
- To prevent including files multiple times, we are going to setup each header file so that it first checks if a unique marco is defined. If it has not yet been defined, we will include the header file and define the marco. If it has been defined, we won't include the file.
- See `includeGuards` directory for this tutorial for an example.

6 Tip of the Week: Preprocessor Error Messages

While programming, we will often want to debug by printing out statements to check if program is doing what we expect it to do. However, if we forget to comment out the print statment afterwards, our program will not do what we expect.

One simple way around this it to wrap this print statements in a preprocessor macro.

```
#ifdef DEBUG
    cerr << "Testing debug mode" << endl;
#endif
```

It can become cumbersome to wrap each statement like this. Another approach is to write a function which will be called for debugging purposes.

```
void debug(const string &s){
    #ifdef DEBUG
        cerr << s << endl;
    #endif
}
```

This function could be overloaded to handle any built in class or struct you define. You will be able to turn on debugging by compiling with the `-DDEBUG` flag.

7 Vi Tip of the Week: Yank, Cut, and Paste

- `y`: copy the current line
- `p`: pastes lines which were yanked or deleted after the cursor
- `P`: pastes lines which were yanked or deleted before the cursor
- Visual Mode: you can press `v` to enter visual mode. This allows you to select text by moving up and down the screen. This text can be yanked using `y` or cut using `x` and pasted using p as described above.

  - you can also use ctrl-v to select a vertical block of code and shift-v to select lines of code.