CS 246 Spring 2018 — Tutorial 4

May 30, 2018

Summary

1 Dynamic Memory

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

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

- To allocate and delete an array on the heap:

```
int *arr = new int[10];
...
delete[] arr;
```

- Like freeing `NULL` in C, deleting a `nullptr` in C++ is safe, so there is no need to check before doing so.
- **Note:** make sure you always use `new` and `delete` together, and use `new[]` and `delete[]` together.

2 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
  - The meanings should be intuitive, e.g. `operator+` should do something that resembles addition.
  - If an operator is overloaded, consider overloading other operators associated with it, e.g. if `operator+` is defined then consider also defining `operator+=`
  - If an operator is commutative, don't forget to define the alternative arrangement (see `lectures/c++/operators/vectors.cc`)

2.1 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.
- In order for cascading to allow modifications to the previously returned value, the result must be returned by reference
- Thus, `operator<<` returns a reference to the ostream (for `operator>>`, returns a reference to the istream) 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.

3 Preprocessor

- The preprocessor runs before the compiler. It handles all preprocessor directives (any line which begins with #).
- Common preprocessor directives:

`#include "file.h"`	inserts the contents of `file.h`
`#define var val`	defines a preprocessor macro `var` with value `val`. If `val` is omitted the value is the empty string

`#ifdef var`	includes following code if `var` is defined. Must be closed with `#endif`
`#ifndef var`	similar to `#ifdef`, but includes code if `var` is not defined

3.1 Include Guards

- As you learnt in CS136, 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 end up including the same file multiple times in our program which will likely result in compilation errors.
- To prevent including the same file multiple times, we are going to setup each header file so that it first checks if a unique macro is defined. If it has not yet been defined, we will define the marco and include the contents. If the macro has been defined, we won't include the contents.
- See `includeGuards` directory for this tutorial for an example.

4 Structures and Classes

- A structure is a collection of data and methods

```
struct Rational {
    int num;
    int den;

    void reduce();
};
```

- To access the fields of an object: `objectName.fieldName`
- Methods are called using `objectName.method()`.
- Fields and methods can be refered using the term **members**.
- There is a pointer called `this` that points to the object the method was called on.
- Methods have access to the members of the object, and members can be accessed directly by calling the member name. We can also access them through `this` but it is often redundant.
- To access members through a pointer, the pointer must be dereferenced:

  `objectName->memberName`

  Note: the above is equivalent to `(*objectName).memberName` but the arrow notation is much cleaner and more readable (especially for code like `a->b->c`).

5 Constructors

- When working with C, when you wanted to program a structure, you would typically write a separate function to allocate memory for the object and initialize the fields to be logical default values.
- In C++, we will instead write constructors. A constructor is a special method which (potentially) initializes the fields of the object.
- Example:

```
struct Vec {
    int x;
    int y;

    Vec(int x, int y): x{x}, y{y} {}
    //                |<--MIL -->|
};
```

- A constructor will always be defined as `ClassName(parameters) {...}`
- Note that we can overload the constructor. In our example above we could also add in `Vec(int x)` if we desire. We can also give the parameters default values.
- A constructor that can be called with no arguments is the **default constructor** for the class. This is the constructor which is called when we have `Vec v;`
- Notice that constructors' **return type is implicit** (i.e. the constructor returns the object constructed, but the type is not in the signature of the function).
- If we do not write a constructor, the compiler usually produces a default constructor and allows C-style struct initialization.
  - The default constructor calls the default constructor for any non-primitive fields and leaves primitive fields uninitialized.
  - If we define our own constructor(s), we lose the implicitly-declared (i.e. compiler provided) default constructor.
  - There are other cases where the implicitly-declared default constructor is lost; can you think of any? Hint: consider the cases where default initialization is not possible.
- The Member Initialization List (MIL) is the **only way to initialize** a variable when the space is allocated to the fields of the object. For some cases it is not required, but you are encouraged to use the MIL as much as possible.
- The following members **need to** be initialized in the MIL:
  - `const` members
  - reference members

  - object members without a default constructor (because when space is allocated, the default constructor will be called, which won't work if there is no default constructor)

- **Note:** When initializing an object, use braces like `Vec vec{1, 2};`, this uses the **uniform initialization syntax**.

6 Tips of the Week: Preprocessor Debug Messages

- While programming, we will often want to debug by using print 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 the 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 Vim Tips of the Week: Repeat Commands, More Motions

- In vim almost all commands can be repeated `n` times by prefixing the command with `n`.
- For example, `2dd` deletes 2 lines, and `4cw` (`c4w` also works) deletes 4 words and enters insert mode.
- A few more ways in normal mode to move around:

  `$` moves to the last character of the line

  `^` moves to the first non-whitespace character of the line

  `%` moves to the matching parenthesis/bracket/brace

  `gg` moves to the first line in the file

  `G` moves to the last line in the file