CS 246 Winter 2014 - Tutorial 6

October 29, 2014

1 Summary

- Valgrind
- Singleton Pattern
- Constructors
- Destructor
- Assignment Operator
- Rule of Three

2 Valgrind

- By now, you've probably realized that Marmoset does not like memory leaks which means you do not like memory leaks.
- However, it can be really hard to determine exactly where memory leaks are occurring and why.
- We have a program which we can use to check for memory leaks - `valgrind`.
- To run valgrind, call

  ```
  valgrind ./exec
  ```

- This runs your program as usual with some extra information being printed to stderr. This extra information tells you if you have memory leaks and if you do a bit of extra information.
- If there is not a memory leak, somewhere you will see the lines

  ```
  HEAP SUMMARY:
      in use at exit: 0 bytes in 0 blocks
  ```

- If there is a memory leak, you should see output similar to

  ```
  HEAP SUMMARY:
      in use at exit: 42 bytes in 7 blocks
  ```

  which means that there are 7 instances where new was called and delete was not called on that data.
- The rest of the complied code from today will be used as example.
- On the topic of memory, you should **NEVER** call `new Class[0]` for any class including ints, chars, strings, or any user defined classes. This gets a block of memory which is 0 bytes. When you call delete on the memory, this memory will not be freed and the block will unfreed. Use `NULL` or `0` to initialize an empty pointer.

3 Singleton Pattern

- Recall, the Singleton design pattern ensures that only a single instance of a class is ever created
- This can be useful when we have shared resource (say, a database)
- So let's implement the Singleton pattern using static (although, we could likely get away using global variables and functions).
- Example: see `Database.cpp` and `Database.h`
- Note the qualified names for function definitions and initial singleton value
- Currently, we still have some problems. People could still create their own Databases as they can access the constructor.
- What if we wanted to set a Database adminstrator what might we need to do?
- When does the destructor get called? Better yet, when do we delete the Database object?
- There are several solutions. In class, you've seen (at least) one. What is it?

4 Visibility

- Sometimes we want to restrict access to methods or fields of an object
- This could be due to privacy concerns or to force a particular usage
- In class, you've seen:
  - `public`: which allows any one to access the field/method
  - `private`: only objects in that class or friends can access the field/method
- Unlike other languages, C++ qualifies `public/private` methods/fields as a section and not on each method or field

```
struct Foo{
 public:
  Foo();
  int getX();
 private:
  int x;
 public:
  Foo(const Foo& f);
 private:
  Foo& operator=(const Foo& f);
};
```

- Recall that we can replace `struct` with `class`
- The only difference is what?
- We can now fix our Database class problem of anyone being able to create an instance using visibility:

```
class Database{
  static Database* singleton;
  unsigned int users;
  Database() : users(0){}
 public:
  static Database* getInstance();
  void addUser(std::string id);
  unsigned int getCount();
};
```

5 Constructors

- Constructors are just special methods that are used to perform intialization immediately following allocation
- Constructors take the name of the class and can be overloaded in the usual fashion
- If we don't define the default constructor (e.g. one that takes no arguments) then the compiler gives us one that does some initialization
  - C++ strings are set to null
  - Sub-objects have their default constructor called
  - Pointers and other primitive data are not initialized
- Basically, the implicit default constructor does enough to make an object valid but not necessarily what we expect
- So we should define constructors ourselves:

```
struct Vector{
  int size;
  int * fields;
  explicit Rational(int size):size(size),fields(new int[size]){
     for (int i = 0; i < size; ++i)
        fields[i] = 0;
  }
};
```

- Once we define any constructor, we lose **every** implicit constructor.

6 Copy Constructor

- The copy constructor is another constructor the compiler will implicitly give us
- It is used to copy an object based upon another object
- Typically, this means that the object being copied should not be changed (and so is a `const` reference)
- Suppose we continue the example of our Vector.
- Then how might we define the copy constructor?

```
struct Vector{
... // Assume other constructors defined correctly
  Vector(const Vector & v): size(v.size), fields(v.fields){}
};
```

- What's the problem? They share fields! That doesn't seem right.
- What we've done is called a **shallow copy**.
- What we really want is a **deep copy**

```
Vector(const Vector & v): size(v.size), fields(new int[size]){
   for (int i = 0; i < size; ++i){
      fields[i] = v.fields[i];
   }
}
```

- Now, the two vectors can have different fields.

7 Destructor

- Destructors are the opposite of Constructors, except you only get one
- A destructor takes the class name, prefixes it with ~, and takes no parameters or return type
- Destructors are used to uninitialize an object at deallocation (e.g. free any heap allocated memory)
- Typically, we use a destructor if we have a non-contiguous object
  - For example, the object has open files, dynamically allocated memory, pointers to other objects, etc
- When is the object's destructor called in the following code:

```
struct Foo{
  int * arr;
  Foo(int n) : arr(new int[n]){}
  ~Foo(){delete [] arr;}
};
int main(){
  Foo x(1);
  Foo y(11);
  Foo *fp = new Foo(20);

  delete fp;
}
```

- What order are the destructors called in? Why this order?

8 Assignment Operator

- Recall, that if we don't define an assignment operator then we get an implict one (like the implicit copy ctor) that does memberwise copy
  - Implicit assignment operator basically performs a shallow copy
- Why might we prefer the copy-and-swap idiom to other methods for defining an assignment operator?
  - Allows implicit garbage collection, we don't have to explicitly delete anything
  - Reuses code from copy constructor - less chance for errors
  - If memory allocation fails, `this` is left in a valid state
- Example: see Vector.cc. To implement the deep copy constructor, destructor and assignment operator, compile with the flag `-DBIGTHREE`
- Why do we return a reference to `*this`?

9 Rule of Three

- The Rule of Three states: *If you define one of copy constructor, destructor, or assignment operator then you likely need all three.*
- Why? Each of this have to be written if the object contains heap allocated data. So if one has to be rewritten, it is likely all of them have to be.
- Do you **always** need to define all three if you define one of them? Why (not)?
  - Not always. Might want to close files, or print logging information in a destructor, etc.