CS 246 Fall 2018 — Tutorial 4

October 12, 2018

Summary

1 Structures and Classes

- A structure is a collection of data and methods

```
struct Complex{
    int real;
    int i;

    int getReal();
    int getImaginary();
};
```

- To access the fields of an object: `objectName.fieldName`
- Methods are called using `objectName.method()`.
- Fields and methods are called **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 usually redundant.
- When is it not redundant to use `this`?
- 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`).

2 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 allocates the memory for a class and (potentially) initializes the fields of the object.
- Example:

```
struct Complex {
    int real;
    int i;

    Complex(int real, int i): real{real}, i{i} {}
    //                        |<---- 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 `Complex(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 `Complex c;`. A class can only have one default constructor. A constructor with all parameters defaulted is a default constructor.
- Notice that a constructor's **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 both the implicitly-declared (i.e. compiler-provided) default constructor and list initialization (for aggregates).
  - There are other cases where the implicitly-declared default constructor is lost; can you think of any? Hint: consider the cases where a "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 **needs to** be initialized in the MIL:
  - `const` members
  - reference members

  - object members without a default constructor

- **Note:** When initializing an object using braces, such as `Vec vec{1, 2};`, the **uniform initialization syntax** is used.

3 Important Topic: Returning from functions

- You may recall from CS136 that you can never return a pointer to an object on the stack. You CAN return an object which is stored on the stack – even in C!
- When returning an object in CS136 from a function, you were taught to allocate the memory for the object on the heap and return the pointer. This leads many students to believe they can only return pointers to objects.
- A benefit of returning by value is that the object will not need to be explicitly deleted by the callee of the function. The stack will take care of deleting it.
- When a function is called, a space in memory is saved between the function calls to save the returned object while the stack of the function is deleted. After the stack of the function is removed, the returned object continue to exists in this temporary location to be used immediately. It can be used as a parameter to a function.
- For instance, consider the code for complex numbers and consider this line:

```
Complex c1{1, -2}, c2{3,5};
cout << c1 + c2 << endl;
```

  We know that the objects c1 and c2 are constructed using the constructor, but what variable is being printed out?

- The object which is calculated as the result of adding c1 and c2 is computed in the stack of operator+. When the object is returned, it is copied into the space between the called and calling functions on the stack. This returned value does not have a permanent location on the stack. It is then used as the parameter to operator«. The exact time this temporary returned value is deleted is not important, but it will be deleted by the time this line of code is complete.
- When do we want to return by reference or pointer? When it is safe to! If the object being returned is going to exist after the function call is done, returning by reference or pointer is safe. For instance, operator+= can return by reference because the object being returned, i.e. this, will exist when the function is done. If the returned object was allocated on the heap, return by pointer.
- Note: you should never allocate an item on the heap and return by reference or value. While the data will continues to exist, it is understood that we do not need to delete references or stack based variables. This will lead to memory leaks or objects being deleted multiple times.