CS 246 Fall 2021 — Tutorial 5

October 6, 2021

Summary

1 Structures and Classes

- A structure is a collection of data and methods.

  ```
  struct Complex{
      int real, imag;

      void init( int real, int imag );
      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 upon which the method was invoked.
- 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 that 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 using 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 that allocates the memory for an instance of a class and initializes the fields of the object.
- A constructor will always be defined as `ClassName(parameters) {...}`
- Note that we can overload the constructor.
- A constructor that can be called with no arguments is the **default constructor** for the class. This is the constructor that is called when we declare `Complex c;`. A class can only have one default constructor. A constructor with all parameters defaulted is still 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 specified in the signature of the function).
- If we do not write a constructor, the compiler 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 where the class doesn't provide a default constructor
- **Note:** When initializing an object using braces, such as `Vec vec{1, 2};`, the *uniform initialization syntax* is used.

3 Destructors

- The *destructor* is a method that is called when a object is destroyed. This is either when it is heap-allocated and `delete` is called on it, or when it goes out of scope.
- A default destructor is provided for us by the compiler. This destructor calls the destructors for all data fields **that are objects**. Note that it will not call `delete` on data fields that are pointers, because pointers are not objects.
- The format of the destructor will always be `~ClassName()`.
- Destructors do not have a return type.

4 Important Topic: Returning from functions

- You may recall from CS136 that you can never return the address of an object on the stack. You CAN return an object that 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 value i.e. the address the pointer contains. This leads many students to believe that 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 either the function or the caller of the function. Instead, the run-time stack will take care of deleting it.
- When a function is called, space in the run-time stack is saved between the function calls into which the returned object is saved while the function stack-frame is deleted. After the function stack-frame is removed, the returned object continues to exist in this temporary location so long as it is used immediately.
- 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?
- When do we want to return by reference or pointer? When it is safe to do so! 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 continue to exist, it is generally understood that we do not need to delete

references or stack-based variables. This will lead to either memory leaks or objects being deleted multiple times.

5 Example: Matrix of floats revisited

Let's re-examine the matrix of floats example from last week now that we have the concept of classes. We would like to write a small C++ program that can read in and output a two-dimensional matrix of real numbers.

Our steps in solving the problem will be similar to those from last week, just using our new techniques.

- Our class `Matrix` still holds the same information as before:

  ```
  struct Matrix {
     unsigned int rows, cols;
     float ** data;
  };
  ```

- We need a way to properly initialize the matrix.
- Let's also define `operator+` for two matrices, that returns a new matrix that is the sum of the two; if the dimensions aren't identical, an empty matrix is instead returned.