CS 246 Spring 2017 - Tutorial 5

June 7, 2017

1 Summary

- valgrind
- Structure and Classes
- Constructors

2 Valgrind

See valgrind/valgrindSlides.pdf in tut5 directory.

3 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**.
- this is a pointer 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. [1]
- To access members through a pointer, the pointer must be dereferenced:

[1] We can also access them through this but it is often redundant.

  - `objectName->memberName`
  - Note: the above is equivalent to `(*objectName).memberName` but this notation is not typically used, and is discouraged.

4 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 Vec{
    int x;
    int y;

    Vec(int x, int y):x{x},y{y} {}
};
```

- 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 with no parameters 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 using this constructor call)**.
- If we do not write a constructor, the compiler produces a default constructor and allows list initialization.
  - The default constructor calls the default constructor for any non-primitive fields (e.g. classes) and leaves primitive types uninitialized.
  - We lose **both the implicitly-declared (i.e. compiler provided) default constructor and list initialization (for aggregates)** if we define our own constructor(s). [2]

[2] There are other cases where the implicitly-declared default constrctor is lost; can you think of any? Hint: consider the cases where a "default initialization" is invalid.

4.1 Constructors for the `Node` class

- **Exercise:** How would we write the constructors in the following class?

```
struct Node{
    int value;
    Node* next = nullptr;
    Node* prev = nullptr;

    Node(int value);          // create a list with one node

    Node(int begin, int end); // create a list with all values from begin
                              // to end in order

    void add(int insert);     // add insert as the last node of the list
};
```

- **Solution**:

```
Node(int value): value{value}{}

Node(int begin, int end): value{begin}{
    int dir = 1;
    if (begin > end){
        dir *= -1;
    }
    while (begin != end){
        begin += dir;
        add(begin);
    }
}

void Node::add(int insert){
    if (next){
        next->add(insert)
    } else {
        next = new Node{insert};
        next->prev = this;
    }
}
```

- The way the methods are currently implemented, we will get a memory leak. Why?

5 Tip of the Week: .vimrc

The ∼/`.vimrc` file contains a list of commands which will run each time `vim` is opened. It can be useful to place commands in the .vimrc file to set user preferences.

Potential useful commands:

- `set number`- turns on line numbers
- `set expandtab`- pressing tab is replaced with spaces
- `set tabwidth=n`- displays and types tabs with width n
- `set shiftwidth=n`- set width of auto-indent tools
- `set smartindent`- indents based on the type of the file

You can also set the colour scheme for vi using your .vimrc file.