Handout: The Big Three: Crafting STL-Like Classes

Big picture: By building and analyzing custom classes that mimic STL components like std::list, you will internalize key C++ design principles—encapsulation, operator overloading, **dynamic memory management** —while appreciating why STL is robust and reusable.

Big Three: Destructor, Copy Constructor, Copy Assignment Operator
C++ provides default versions of all three functions, but you need to write your own if your class uses dynamic memory.

`Complex` Class (No Dynamic Memory)

Here's the updated `Complex` class with no dynamic memory from last lecture. The default big three work perfectly fine!

```C/C++
class Complex {
public:
    Complex(double re = 0.0, double im = 0.0) {
        real = re;
        imag = im;
    }
    double getReal() const { return real; }
    double getImag() const { return imag; }
    void setReal(double re) { real = re; }
    void print() const {
        cout << real << " + "
             << imag << "j" << endl;
    }

private:
    double real;
    double imag;
};
```

`Complex` Class (With Dynamic Memory)

Here's the updated `Complex` class with dynamic memory. We'll explore why it needs special handling.

```C/C++
class Complex {
public:
    Complex(double re = 0.0, double im = 0.0) {
        data = new double[2];  // Dynamic array
        data[0] = re;          // Real part
        data[1] = im;          // Imaginary part
    }
    double getReal() const { return data[0]; }
    double getImag() const { return data[1]; }
    void setReal(double re) { data[0] = re; }
    void print() const {
        cout << data[0] << " + "
             << data[1] << "j" << endl;
    }
    // Big Three methods (live coding)









private:
    double* data;  // Pointer to dynamic memory
};
```

Activity 1: Memory Diagrams with Defaults

Goal: Understand how default behavior works (or fails) with both versions of `Complex`. Draw memory diagrams for the stack and heap (if applicable). Use arrows for pointers and boxes for objects/values.

```C/C++
int foo() {
    Complex x(3.0, 4.0);
    Complex y = x;  // Default copy constructor
    y.setReal(5.0);
    x.print();
    y.print();
    return 0;
}  // Objects destroyed here
```

1. Draw the stack and heap after the first three lines are executed.

Assume Complex class that uses dynamic memory (on page 2)

2. Will the values printed for x and y differ (as intended in the code) ?

A. Yes B. No

3. What happens when `x` and `y` are destroyed after `foo()` returns?
Hint: No destructor yet!

A. No problem occurs
B. Something goes wrong at **compile time**
C. Something goes wrong at **run time**
(Memory Leaks / Crashes /Double deletion/incorrect output)

Note: To check for memory leaks, run your executable (a.out) through the tool valgrind. See usage below:

```
Unset
valgrind --leak-check=full ./a.out
```

(10 mins) Activity 2: Fixing issues in `Complex`
Goal: Predict how implementing the Big Three solves the problems. You'll see these implemented live!

1. **Destructor**: Add `~Complex()`. Use `delete[] data;` to free the array. After adding `~Complex()` to free data, does it fix everything?
   A. Yes, all memory issues are fixed.
   B. No, copies still share the same data.
   C. No, it causes a crash immediately.
   D. Yes, but only for one object.

Note: Use gdb to step through the code and see how execution jumps into the destructor function (`~Complex()`), right after `foo()` returns.

gdb commands (today)[1]

- gdb <name of executable> //Start GDB and load the executable
- [b] or break <filename:line number> //Set a breakpoint on a specific line
- [r] or run //Run the code until you hit the first breakpoint
- [n] or next //Execute the next line of code
- [s] or step //Step into a function
- [bt] or backtrace //Show call stack

[1] For more commands see this gdb cheatsheet: https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

2. **Copy Constructor**: What does a proper copy constructor, `Complex(const Complex& other)`, need to do to avoid sharing? Draw the stack and heap for `Complex y = x` with a proper deep copy.

    A. Copy the pointer value directly (`data = other.data`)
    B. Allocate new memory and copy the values
    ```
    data = new double[2];
    data[0] = other.data[0];
    ...
    ```
    C. Set `data` to `nullptr`
    D. Free `other.data` before copying

3. **Copy Assignment Operator**: For `y = x` (where `y` already exists), what must `operator=` do with `y`'s old `data`?

```C/C++
int foo() {
    Complex x(3.0, 4.0); // Create x
    Complex y(1.0, 2.0); // Create y with
                         // different value
    y = x; //Calls copy-assignment operator
    y.setReal(5.0);
    x.print();      // ?
    y.print();      // ?
    return 0;
}  // Objects destroyed here
```

Draw the heap before and after the assignment (`y = x;`). Do you see any issues if the default assignment operator were used?

Rule of Three in C++ states that if a class defines one (or more) of the Big three special member functions, it should explicitly define all three to ensure proper resource management.