

CSE 333 Section 4 - C++ Classes, Dynamic Memory

Welcome back to section! We're glad that you're here :)

Member, Non-Member, and Friends, Oh My!

Exercise 1) Complete the following table (use objects obj1 and obj2, where appropriate):

| | Member | Non-member |
|----------------------------|--------|------------|
| Access to Private Members: | | |
| Function call (Func): | | |
| Operator call (*): | | |
| When preferred: | | |

Constructors, Destructors, what is going on?

- **Constructor:** Can define any number as long as they have different parameters. Constructs a new instance of the class. The *default constructor* takes no arguments.
- **Copy Constructor:** Creates a new instance of the class based on another instance (it's the constructor that takes a reference to an object of the same class). Automatically invoked when passing or returning a non-reference object to/from a function.
- **Assignment Operator:** Assigns the values of the right-hand-expression to the left-hand-side instance.
- **Destructor:** Cleans up the class instance, e.g., free dynamically allocated memory used by this class instance.

What happens if you don't define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad?

How can you disable the copy constructor/assignment operator/destructor?

Exercise 2) Order the execution of the following program:

```
class Bar {
public:
    Bar() : num_(0) { } // 0-arg ctor
    Bar(int num) : num_(num) { } // 1-arg ctor
    Bar(const Bar& other) : num_(other.num_) { } // cctor
    ~Bar() { } // dtor
    Bar& operator=(const Bar& other) = default; // op=
    int get_num() const { return num_; } // getter

private:
    int num_;
};
```

```
class Foo {
public:
    Foo() : bar_(5) { } // 0-arg ctor
    Foo(const Bar& b) { bar_ = b; } // 1-arg ctor
    ~Foo() { } // dtor

private:
    Bar bar_;
};
```

```
int main() {
    Bar b1(3);
    Bar b2 = b1;
    Foo f1;
    Foo f2(b2);
    return EXIT_SUCCESS;
}
```

Number the following starting with 1.

Each method may be called more than once (i.e., you can put multiple numbers on the same line).

```
_____ Bar 0-arg ctor
_____ Bar 1-arg ctor
_____ Bar cctor
_____ Bar op=
_____ Foo 0-arg ctor
_____ Foo 1-arg ctor
_____ Foo dtor
_____ Bar dtor
```

Dynamically-Allocated Memory: New and Delete

In C++, memory can be heap-allocated using the keywords “new” and “delete”. You can think of these like `malloc()` and `free()` with some key differences:

- Unlike `malloc()` and `free()`, `new` and `delete` are operators, not functions.
- The implementation of allocating heap space may vary between `malloc` and `new`.

New: Allocates the type on the heap, calling the specified constructor if it is a class type. Syntax for arrays is “`new type[num]`”. Returns a pointer to the type.

Delete: Deallocates the type from the heap, calling the destructor if it is a class type. For anything you called “new” on, you should at some point call “delete” to clean it up. Syntax for arrays is “`delete[] name`”.

Just like baking soda and vinegar, you shouldn't mix `malloc/free` with `new/delete`.

Exercise 3) Memory Leaks

```
class Leaky {
public:
    Leaky() { x_ = new int(5); }
private:
    int* x_;
};

int main(int argc, char** argv) {
    Leaky** dbl_ptr = new Leaky*;
    Leaky* lky_ptr = new Leaky();
    *dbl_ptr = lky_ptr;
    delete dbl_ptr;
    return EXIT_SUCCESS;
}
```

What is leaked by this program? How would you fix the memory leaks?

Exercise 4) Identify the memory error with the following code.

```
class BadCopy {
public:
    BadCopy() { arr_ = new int[5]; }
    ~BadCopy() { delete [] arr_; }
private:
    int* arr_;
};

int main(int argc, char** argv) {
    BadCopy* bc1 = new BadCopy;
    BadCopy* bc2 = new BadCopy(*bc1); // cctor

    delete bc1;
    delete bc2;

    return EXIT_SUCCESS;
}
```

Hint: Draw a memory diagram. What happens when bc1 gets deleted?

Exercise 5) Classes usage [Extra Practice]. Consider the following classes:

```
class IntArrayList {
public:
    IntArrayList()
        : array_(new int[MAXSIZE]), len_(0), maxsize_(MAXSIZE) { }
    IntArrayList(const int* const arr, size_t len)
        : len_(len), maxsize_(len*2) {
        array_ = new int[maxsize_];
        memcpy(array_, arr, len * sizeof(int));
    }

    IntArrayList(const IntArrayList& rhs) {
        len_ = rhs.len_;
        maxsize_ = rhs.maxsize_;
        array_ = new int[maxsize_];
        memcpy(array_, rhs.array_, maxsize_ * sizeof(int));
    }
    // synthesized destructor
    // synthesized assignment operator

private:
    int* array_;
    size_t len_;
    size_t maxsize_;
};

class Wrap {
public:
    Wrap() : p_(nullptr) {}
    Wrap(IntArrayList* p) : p_(p) { *p_ = *p; }
    IntArrayList* p() const { return p_; }
private:
    IntArrayList* p_;
};

struct List {
    IntArrayList v;
};
```

Here's an example program using these classes:

```
int main(int argc, char** argv) {
    IntArrayList a;
    IntArrayList* b = new IntArrayList();
    struct List l { a };
    struct List m { *b };
    Wrap w(b);
    delete b;
    return EXIT_SUCCESS;
}
```

Draw a memory diagram of the program:

How does the above program leak memory?

Fix the issue in the code above. You may write the solution here.