

IICSE 333 Section 5 - 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:

	Member	Non-member
Access to Private Members:	Always	<ul style="list-style-type: none">• Through getters and setters• Through friend keyword (do not use unless needed)
Function call (Func):	<code>obj1.Func(obj2)</code>	<code>Func(obj1, obj2)</code>
Operator call (*):	<code>obj1 * obj2</code>	<code>obj1 * obj2</code>
When preferred:	<ul style="list-style-type: none">• Functions that <i>mutate</i> the object• "Core" class functionality	<ul style="list-style-type: none">• <i>Non-mutating</i> functions• Commutative functions• When the class must be on the right-hand side

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, *i.e.* 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? (Hint: What if a member of a class is a pointer to a heap-allocated struct?)

In C++, if you don't define any of these, a default one will be synthesized for you.

- The synthesized copy constructor does a shallow copy of all fields.
- The synthesized assignment operator does a shallow copy of all fields.
- The synthesized destructor calls the destructors of any fields that have them.

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

Set their prototypes equal to the keyword "delete": `~SomeClass() = delete;`

When is the initialization list of a constructor run, and in what order are data members initialized?

The initialization list is run before the body of the ctor, and data members are initialized in the order that they are defined in the class, not by initialization list ordering

What happens if data members are not included in the initialization list?

Data members that don't appear in the initialization list are *default initialized/constructed* before the ctor body is executed. Including when there is **no** initialization list!

Constructor/Destructor Ordering

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).

```
6 _____ Bar 0-arg ctor
1,4 _____ Bar 1-arg ctor
2 _____ Bar cctor
7 _____ Bar op=
3 _____ Foo 0-arg ctor
5 _____ Foo 1-arg ctor
```

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

```
#include <cstdlib>

class Leaky {
public:
    Leaky() { x_ = new int(5); }
    ~Leaky() { delete x_; } // Delete the allocated int
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;
    delete lky_ptr; // Delete of dbl_ptr doesn't delete what lky_ptr
points to
    return EXIT_SUCCESS;
}
```

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

Deleting the `dbl_ptr` doesn’t automatically delete what the pointer points to. Have to also delete `lky_ptr` and then create a destructor that deletes the allocated int pointer `x_`.

Exercise 4) Identify the memory error with the following code. Then fix it! [Extra Practice]

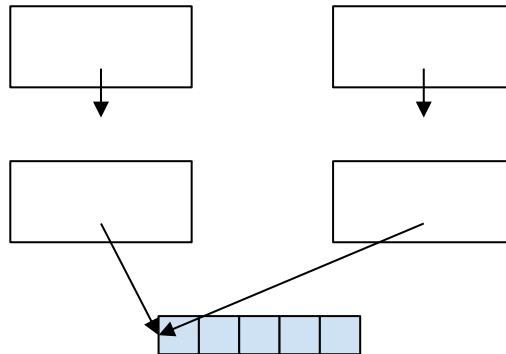
```
class BadCopy {
public:
    BadCopy() { arr_ = new int[5]; }
    ~BadCopy() { delete [] arr_; }
private:
    int* arr_;
};

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

    delete bc1;
    delete bc2;

    return EXIT_SUCCESS;
}
```

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



The default copy constructor does a shallow copy of the fields, so `bc2`'s `arr_` points to the same array as `bc1`'s `arr_`. When `bc1` gets deleted, so does its `arr_`. But this `arr_` is the same one `bc2`'s `arr_` points to, so when `bc2` gets deleted, its `arr_` has already been deleted, leading to an invalid delete (similar to a double `free()`).

5) Classes usage. 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_;
};

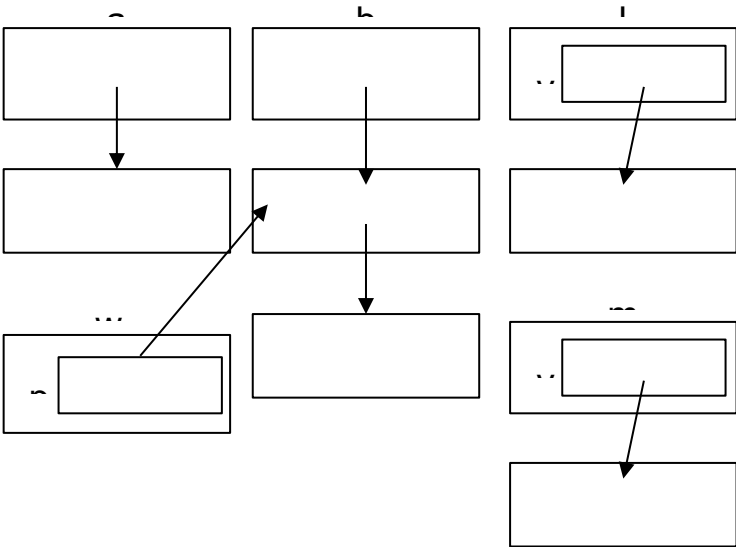
class Wrap {
public:
    Wrap() : p_(nullptr) {}
    Wrap(IntArrayList* p) : p_(p) { *p_ = *p; }
    IntArrayList* p() const { return p_; }
private:
    IntArrayList* p_;
};

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?

The synthesized destructor does not know how to delete an array, so IntArrayList a will leak. Similarly, synthesized destructor does not know how to delete b's array, so IntArrayList* b will leak. struct List l copies a's contents using the copy constructor, and when it gets deleted it calls IntArrayList's destructor, which doesn't know how to delete an array, so this will leak too. struct List m copies what b points to into its own field using the copy constructor, when it gets deleted it does the same thing as struct List l and leaks. Wrap w just copies the pointer, and the synthesized assignment operator shallow copies the fields, so w just points to what b points to through its field p_.

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

Implement the destructor:

```
IntArrayList::~~IntArrayList() { delete[] array_; }
```

Extra Practice - Past Midterm Question

Consider the following (very unusual) C++ program which does compile and execute successfully. Write the output produced when it is executed.

```
#include <iostream>
using namespace std;

class foo {
public:
    foo()                { cout << "p"; }           // ctor
    foo(int i)           { cout << "a"; }           // ctor (1 int)
    foo(int i, int j)    { cout << "h"; }           // ctor (2
ints)
    ~foo()               { cout << "s"; }           // dtor
};

class bar {
public:
    bar(): foo_(new foo()) { cout << "g"; }           // ctor
    bar(int i): foo_(new foo(i)) { cout << "p"; }       // ctor (1 int)
    ~bar()                 { cout << "e"; delete foo_; } // dtor
private:
    foo *foo_;
    foo otherfoo_;
};

class baz {
public:
    baz(int a,int b,int c) : bar_(a), foo_(b,c)
                                { cout << "i"; }           // ctor (3
ints)
    ~baz()                     { cout << "n"; }           // dtor
private:
    foo foo_;
    bar bar_;
};

int main() {
    baz b(1,2,3);
    return EXIT_SUCCESS;
}
```

"happiness" (yes, with 3 s's):

Constructing `b` constructs `foo_(2,3)` first [**h**], then `bar_(1)`, which initializes `foo_` (a pointer, not an object) to `new foo(1)` [**a**] and default constructs `otherfoo_` [**p**] before printing [**p**]. The body of `b`'s constructor then prints [**i**]. As we exit from `main`, `b` destructs, which runs the destructor body [**n**] before destructing `bar_`, which prints [**e**] before deleting the unnamed `foo(1)` [**s**] pointed to by `foo_` and then destructing `otherfoo_` [**s**]. Finally, `foo_` in `b` is destructed [**s**].