

# CSE 333 Section 5 - C++ Classes, Dynamic Memory

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

## **Quick Class Review:**

What do the following modifiers mean?

- public:
- protected:
- private:
- friend:

What is the default access modifier for a struct in C++?

## **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?

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

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

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

**Exercise 1) Give the output of the following program:**

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

class Int {
public:
    Int() { ival_ = 17; cout << "default(" << ival_ << ")" << endl; }

    Int(int n) { ival_ = n; cout << "ctor(" << ival_ << ")" << endl; }

    Int(const Int &n) {
        ival_ = n.ival_;
        cout << "cctor(" << ival_ << ")" << endl;
    }

    ~Int() { cout << "dtor(" << ival_ << ")" << endl; }

    int get() const {
        cout << "get(" << ival_ << ")" << endl;
        return ival_;
    }

    void set(int n) {
        ival_ = n;
        cout << "set(" << ival_ << ")" << endl;
    }

private:
    int ival_;
};

int main(int argc, char **argv) {
    Int p;
    Int q(p);
    Int r(5);
    q.set(p.get()+1);
    return EXIT_SUCCESS;
}
```

## ***Object Construction and Initialization*** **Exercise 2)**

```
#include <iostream>

using namespace std;

class Foo {
public:
    Foo()      { cout << 'u'; }
    Foo(int x) { cout << 'n'; }
    ~Foo()     { cout << 'd'; }
};

class Bar {
public:
    Bar(int x) { other_ = new Foo(x); cout << 'g'; }
    ~Bar()     { delete other_;      cout << 'e'; }
private:
    Foo* other_;
};

class Baz {
public:
    Baz(int z) : bar_(z) { cout << 'r'; }
    ~Baz()             { cout << 'a'; }
private:
    Foo foo_;
    Bar bar_;
};

int main(){
    Baz (1);
    cout << endl; // to flush the buffer
}
```

## ***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); }
private:
    int* x_;
};

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

Draw a memory diagram of the program.

Assuming an instance of `Leaky` takes up 8 bytes (like a C-struct with just `int* x_`), how many bytes of memory are 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); // BadCopy's ctor

    delete bc1;
    delete bc2;

    return EXIT_SUCCESS;
}
```

Draw a memory diagram of the program.

What happens when bc1 gets deleted?

**Exercise 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?

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

## 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.

Hints: Member variables are initialized in declaration order. Destruction order is the reverse of construction order. The body of a constructor runs after its initializer list.

```
#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;
}
```