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`: Member is accessible by anyone
- `protected`: Member is accessible by this class and any derived classes.
- `private`: Member is only accessible by this class

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

A struct can be thought of as a class where all members are default public instead of default private. In C++, it is also possible to have member functions (such as a constructor) in structs.

Constructors, Destructors, what is going on?

What happens if you don't define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad? (<u>Hint</u>: 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?

Use 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!

Exercise 1) Give one possible output of the following program:

```cpp
#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);
  Int s = r;
  q.set(p.get()+1);
  return EXIT_SUCCESS;
}
```

```
   default(17)
   cctor(17)
   ctor(5)
   cctor(5)
   get(17)
   set(18)
   dtor(5)
   dtor(5)
   dtor(18)
   dtor(17)
```

Exercise 3) Identify (and fix) any issues with this HeapInt class.

```cpp
#include <cstdlib>

class HeapInt {
 public:
  HeapInt() { x_ = new int(5); }
  ~HeapInt() { delete x_; }  // Delete the allocated int
 private:
  int* x_;
};

int main(int argc, char** argv) {
  HeapInt** heap_int_ptr = new HeapInt*;
  HeapInt* heap_int = new HeapInt();
  *heap_int_ptr = heap_int;
  delete heap_int_ptr;
  // Deleting heap_int_ptr doesn't delete what heap_int points to
  delete heap_int;
  return EXIT_SUCCESS;
}
```

Assuming an instance of `HeapInt` takes up 8 bytes (like a C-struct with just `int* x_`), how many bytes of memory are leaked by this program (if any)? How would you fix the memory leaks?

Leaks 12 bytes of memory: 8 bytes for the allocated `HeapInt` object `heap_int` points to + 4 bytes for the `int` the Leaky instance allocates in its constructor.

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

Exercise 4) Identify any errors with the following code. Then fix them!

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

int main(int argc, char** argv) {
   IntArr* i_ar1 = new IntArr;
   IntArr* i_ar2 = new IntArr(*i_ar1);
   delete i_ar1;
   delete i_ar2;
   return EXIT_SUCCESS;
}
```

Draw a memory diagram. What happens when `i_ar1` gets deleted?
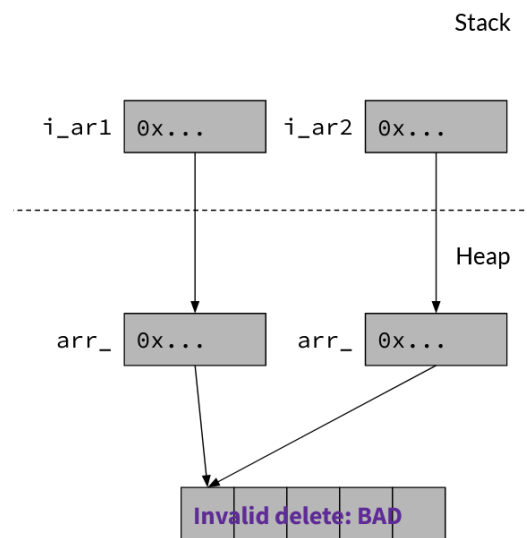
The default copy constructor does a shallow copy of the fields, so `i_ar2`'s `arr_` points to the same array as `i_ar1`'s `arr_`. When `i_ar1` gets deleted, so does its `arr_`. But this `arr_` is the same one `i_ar2`'s `arr_` points to, so when `i_ar2` gets deleted, its `arr_` has already been deleted, leading to an invalid delete (similar to a double `free()`).

Bonus 1) Give the output of the following code

```cpp
#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
}
```

Constructing b as Baz(1) in `main` default constructs `foo_` **[u]** since it is declared first, then constructs `bar_(1)` which runs `Foo(1)` **[n]** and then runs its body **[g]**. We now run the ctor body of `Baz` **[r]**. As we exit from `main`, b destructs, which runs the destructor body **[a]**, then destructs `bar_`, which calls delete on its `Foo*` member **[d]** before printing **[e]**, then we destruct b's `foo_` **[d]**.

Bonus 2) Class usage. Consider the following classes:

```cpp
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;
};
```
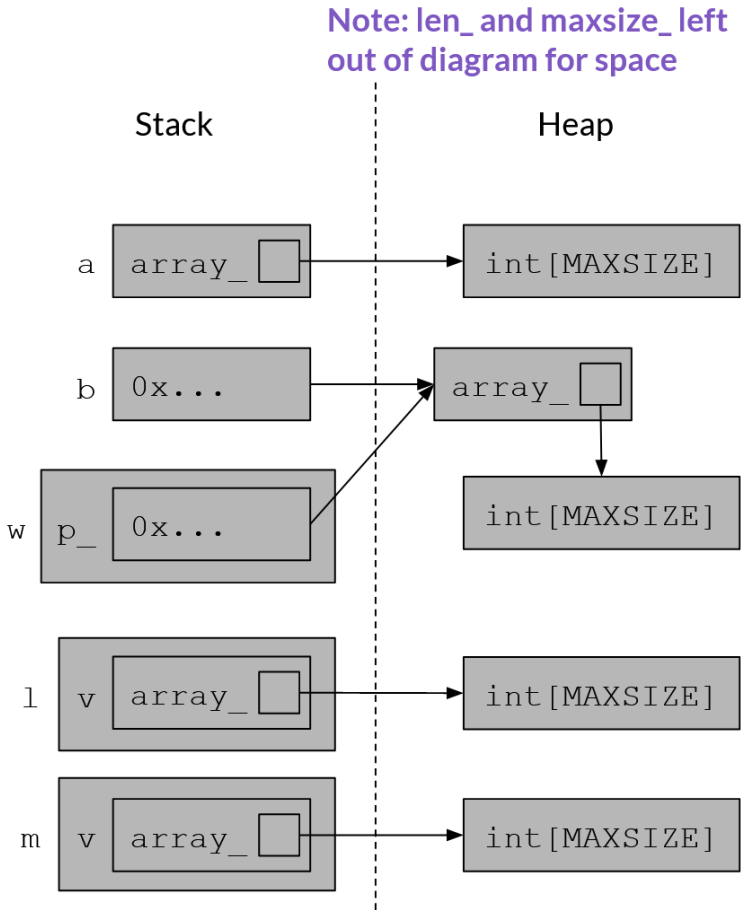
Here's an example program using these classes:

```
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:

Note: len_ and maxsize_ left
out of diagram for space

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_; }
```

Bonus 3) 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;
}
```

"happinesss" (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]**.