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
- `friend`: Allows access of private and protected members to other functions and/or classes

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 give member functions (such as a constructor) to structs.

What happens when we assign an instance of a struct that contains an array to another instance of the same struct?

The compiler will automatically copy each array element into the the other struct instance.

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;`

Exercise:

1) **Give one possible 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;
}
```

```
1. default(17)
2. cctor(17)
3. ctor(5)
4. get(17)
5. set(18)
6. dtor(5)
```

7. dtor(18)
8. dtor(17)

Object Construction and Initialization

Exercise:

2) 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**].

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.

Exercises:

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** lkyptr = new Leaky*;
  Leaky* lky = new Leaky();
  *lkyptr = lky;
  delete lkyptr;
  delete lky;  // Delete of lkyptr doesn't delete what lky points
to
  return EXIT_SUCCESS;
}
```

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?

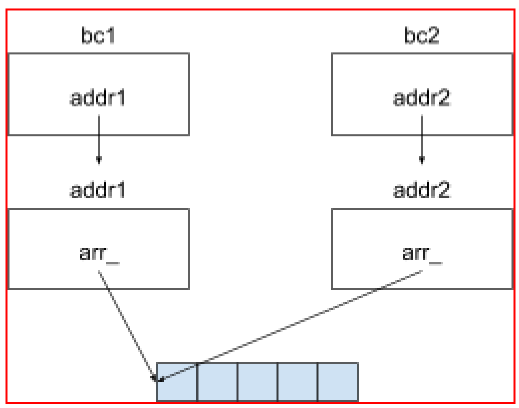
Leaks 12 bytes of memory: 8 bytes for the allocated Leaky object `lky` points to + 4 bytes for the `int` the Leaky instance allocates in its constructor.

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

4) Identify the memory error with the following code. Then fix it!

```
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 cctor

  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()`).

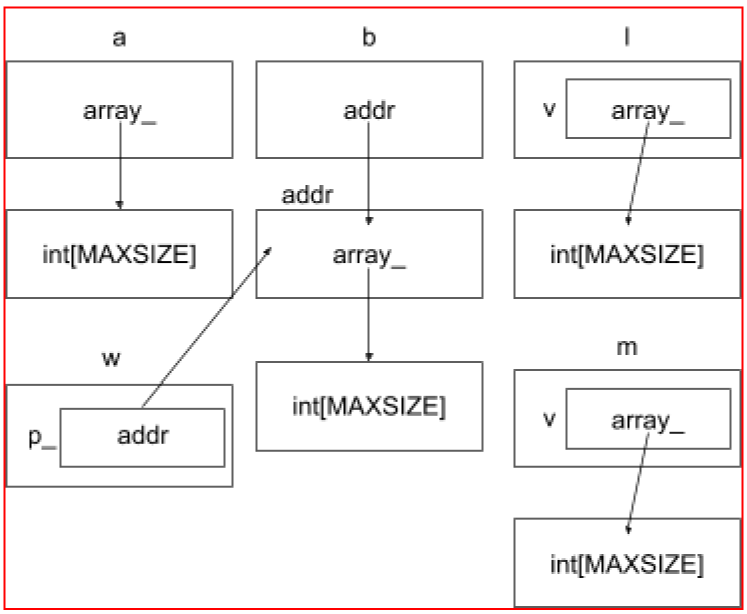
5) Classes usage. Consider the following classes:

```
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_; }
```