CSE 333 Section 5

C++ Classes and Dynamic Memory

Logistics

Due Today:

Homework 2 @ **11:59 pm**

Due Wed (02/09):

Exercise 7 @ **11 am**

Review Questions

- What do the following access 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/protected members to *foreign* functions and/or classes where this modifier is present

- 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 a `struct`

Review: Member vs Non-Member

Member	Non-member
• Used when modifying the object (reassigning and accessing data members) • "Core" class functionality	• Used for **non-modifying** and/or commutative functions. • When operating with the class on the right-hand side
• Allows access to private functions/data members	• Does **NOT** give access to private functions/data members • Only give `friend` keyword if **NEEDED** ○ `friend` allows for non-member private access
• **Function call:** `obj1.Function(obj2);` • **Operator Overloads:** `obj1 *= obj2;`	• **Function call:** `Func(obj1, obj2);` • **Operator Overloads:** `obj1 * obj2;`

Constructors Revisited

```
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; }
};
```

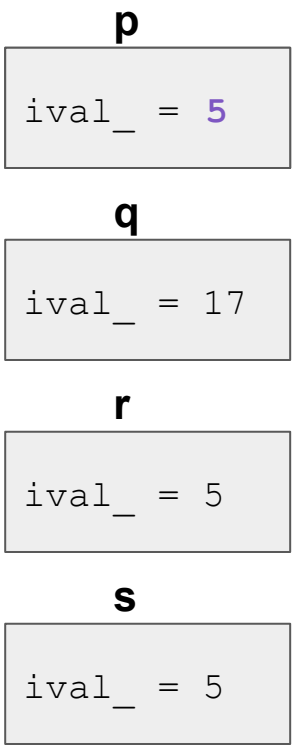
Constructor (ctor): Can define any number as long as they have different parameters. Constructs a new instance of the class.

Copy Constructor (cctor): Creates a new instance based on another instance (must take a reference!). Invoked when passing/returning a **non-reference** object to/from a function.

Destructor (dtor): Cleans up the class instance. Deletes dynamically allocated memory (if any).

What is getting called here?

```
int main(int argc, char** argv) {
  Int p;          // 1. default ctor
  Int q(p);       // 2. copy ctor
  Int r(5);       // 3. 1 arg ctor
  Int s = r;      // 4. copy ctor
  p = s;          // 5. assignment operator
}
```



Initialization Lists

- 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?

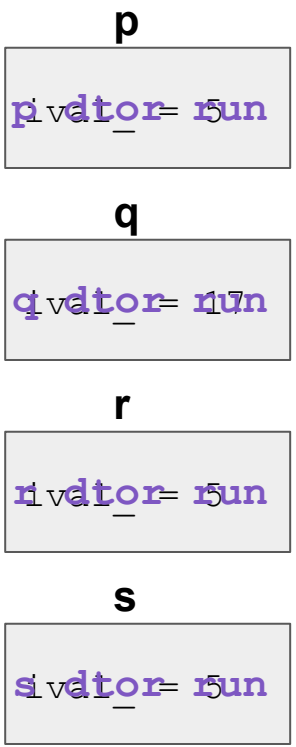
  Data members that don't appear in the initialization list are *default initialized/constructed* before ctor body is executed.

Destructors Review

- When are destructors invoked? In what order are they invoked when multiple objects are getting destructed?
  - An object's destructor is run when it falls out of scope, or when the `delete` keyword is used on heap allocated objects constructed with `new`
  - Invoked in reverse order of construction
- What happens when a destructor actually executes? (Hint: what happens if a dtor body doesn't destruct all of its members?)
  - Destructors are run in reverse order of construction: (1) run destructor body (2) destruct remaining members in reverse order of declaration

When are these destructors run?

```
int main(int argc, char** argv) {
  Int p;
  Int q(p);
  Int r(5);
  Int s = r;
  p = s;
}
```



Steps for Construction and Destruction

Construction:

1. Construct/initialize members in order of declaration:
    - If: member appears in initialization list, apply initialization
    - Else: default initialize
2. Run constructor body

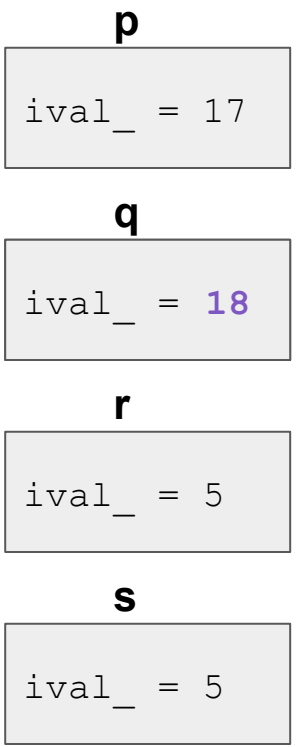
Destruction:

1. Run destructor body
2. Destruct remaining members in reverse order of member declaration

Exercise 1: Constructors and Destructors

```
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;
}
```

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



Design Considerations

- What happens if you don't define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad?
  - In C++, if you don't define any of these, 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 default 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;`

New and Delete Operators

New: Allocates the type on the heap, calling specified constructor if it is a class type

Syntax:

```
type* ptr = new type;

type* heap_arr = new type[num];
```

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:

```
delete ptr;

delete[] heap_arr;
```

Exercise 3: Memory Leaks

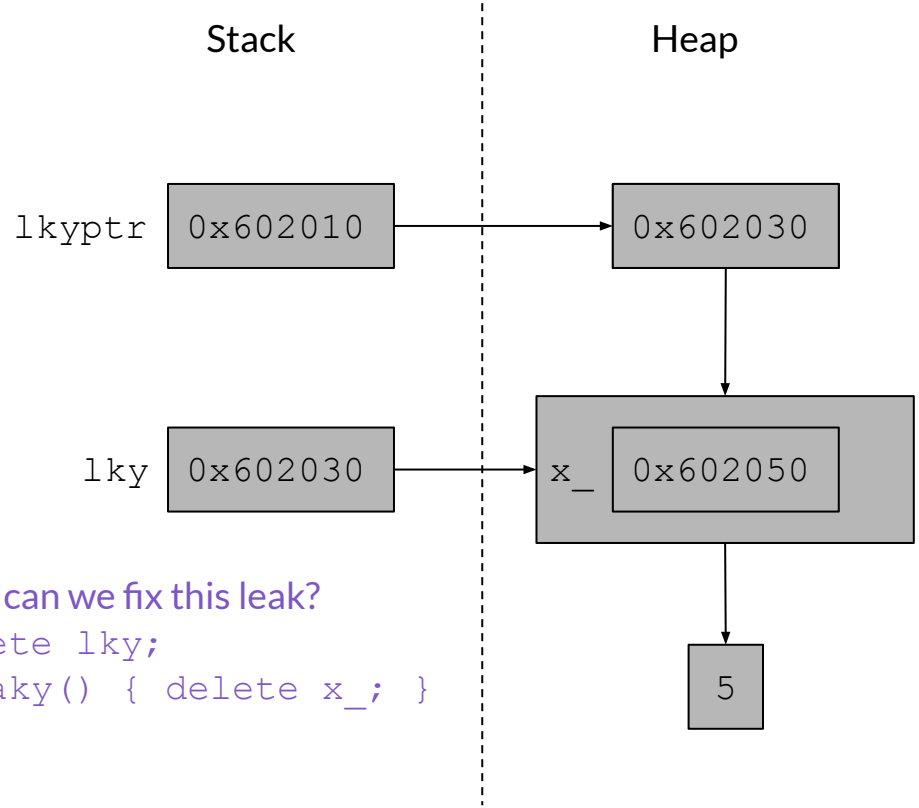
Stack

Heap

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

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

Exercise 3: Memory Leaks

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

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



How can we fix this leak?

```
delete lky;
~Leaky() { delete x_; }
```

Exercise 4: Bad Copy

Stack

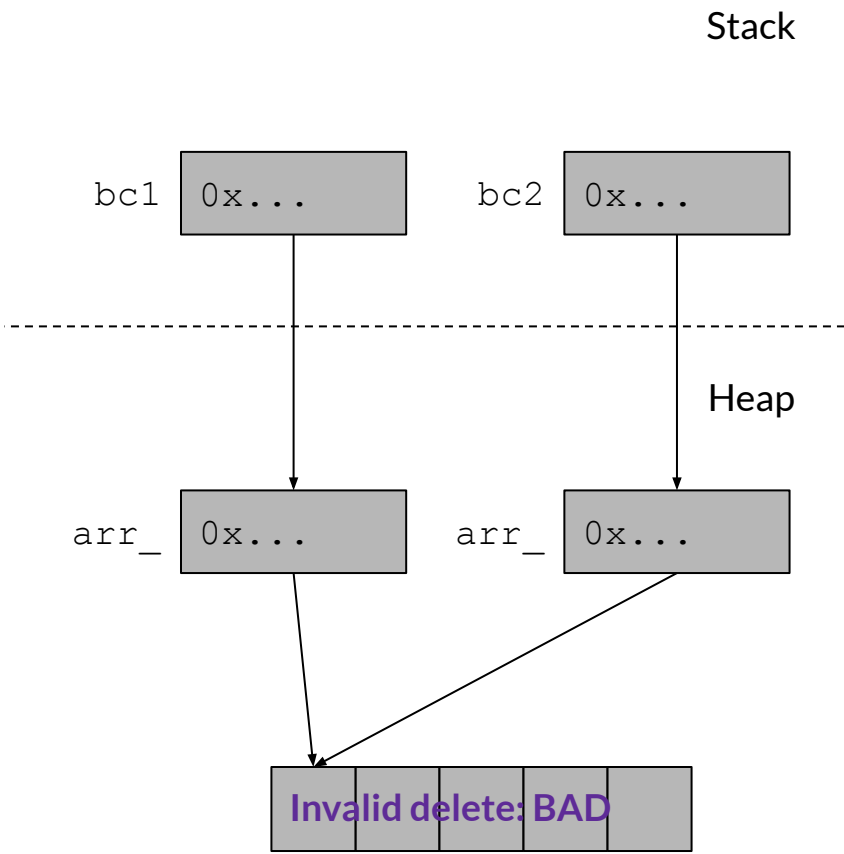
Heap

```
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;
}
```

Exercise 4: Bad Copy

```
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);
  delete bc1;
  delete bc2;
  return EXIT_SUCCESS;
}
```

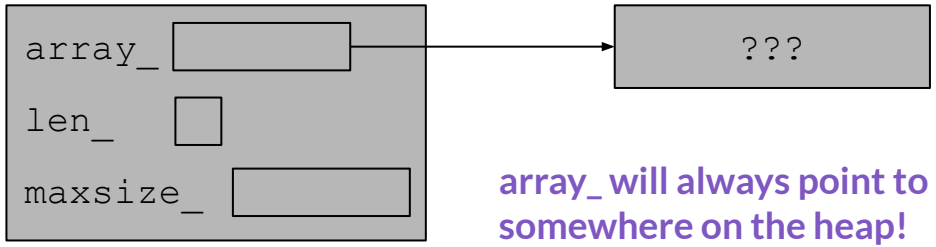
as if!



Exercise 5: IntArrayList

Note: Implementation details on the worksheet

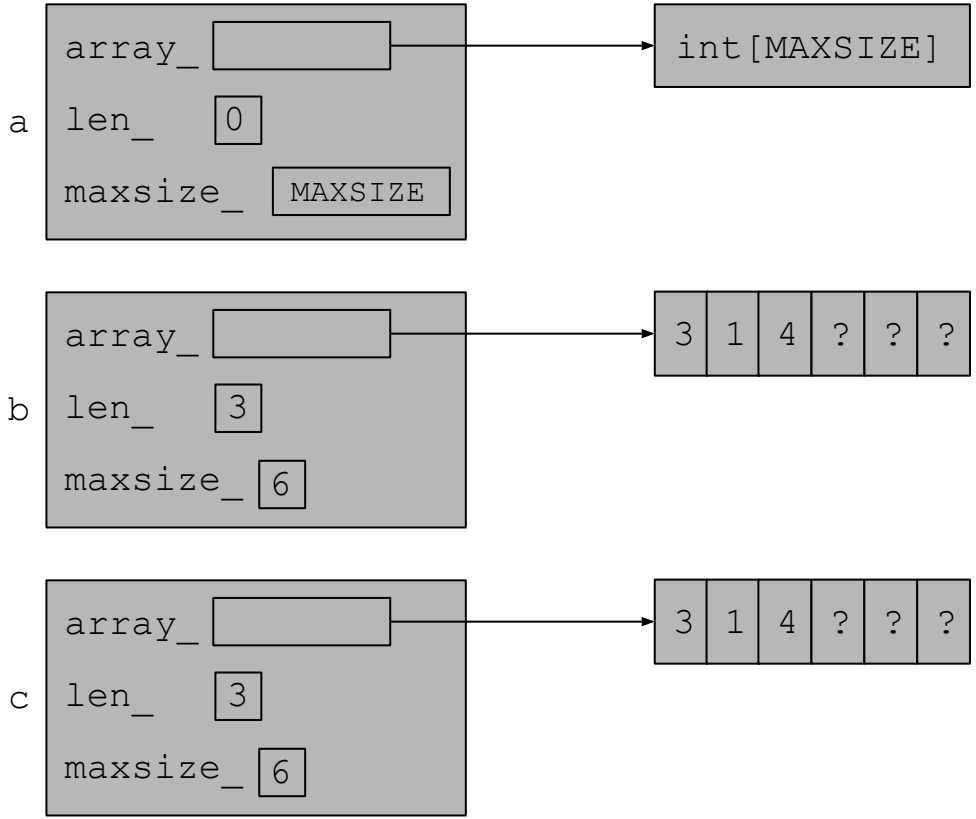
```
class IntArrayList {
  public:
    IntArrayList();
    IntArrayList(const int* const arr, size_t len);
    IntArrayList(const IntArrayList &rhs);
    // synthesized destructor
    // synthesized assignment operator

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



array_ will always point to somewhere on the heap!

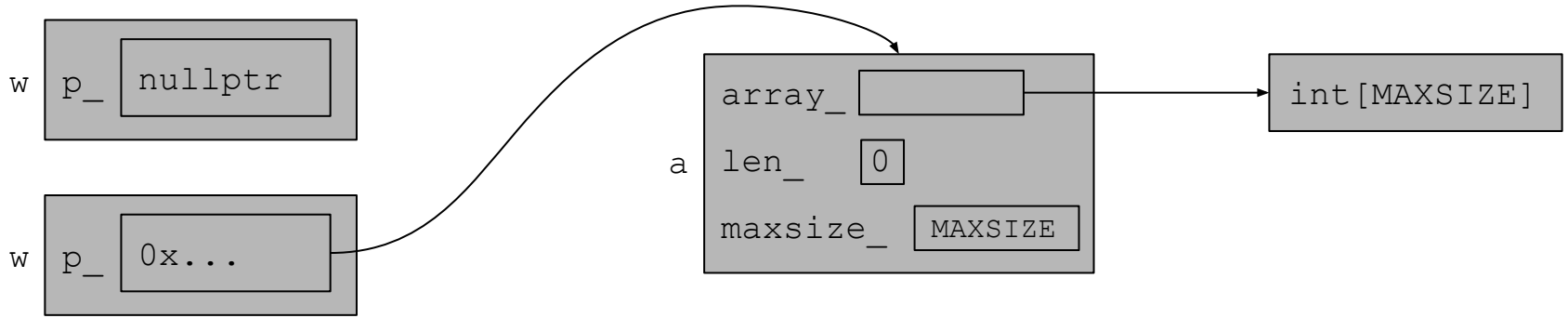
Exercise 5: IntArrayList

```
int main() {
  IntArrayList a;
  int copy_me[3] = {3,1,4};
  IntArrayList b(copy_me,3);
  IntArrayList c(b);
}
```



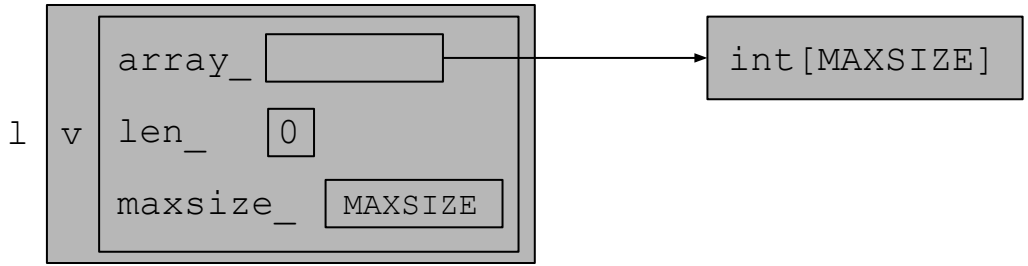
Exercise 5: Wrap

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



Exercise 5: struct List

```
struct List {
    IntArrayList v;
};
```



Exercise 5: Classes Usage

Stack

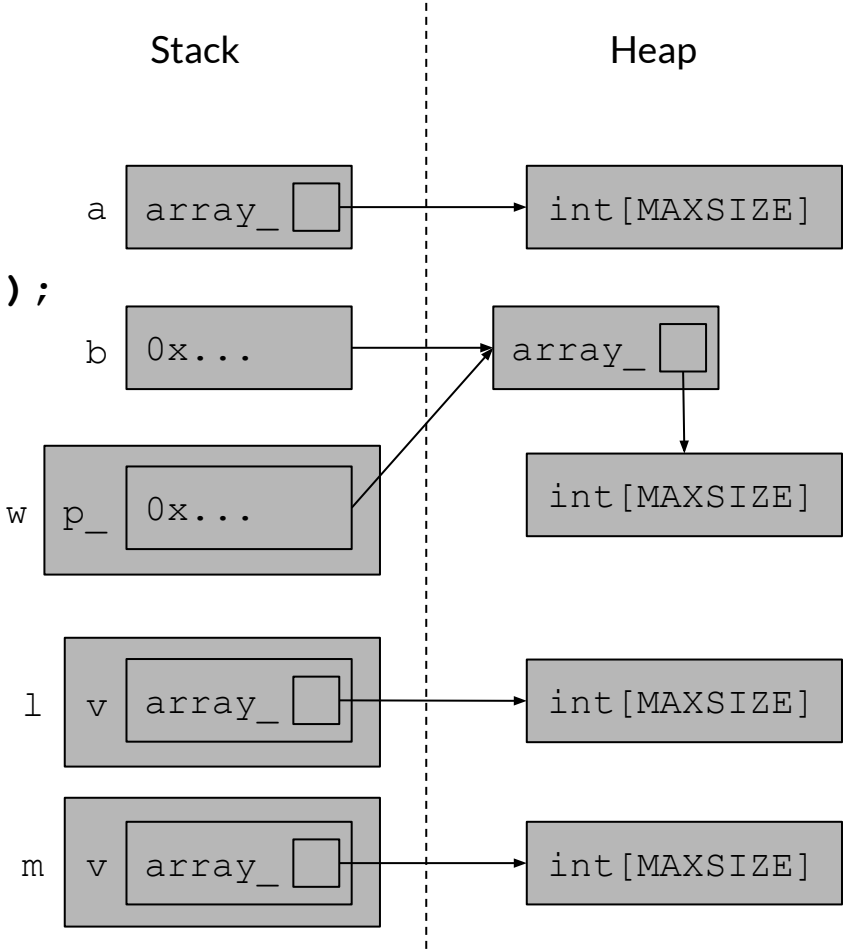
Heap

```
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;
}
```

Exercise 5: Classes Usage

```
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;
}
```

Note: len_ and maxsize_ left out of diagram for space



Exercise 5: Classes Usage

```
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;
}
```

Implement the destructor:

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