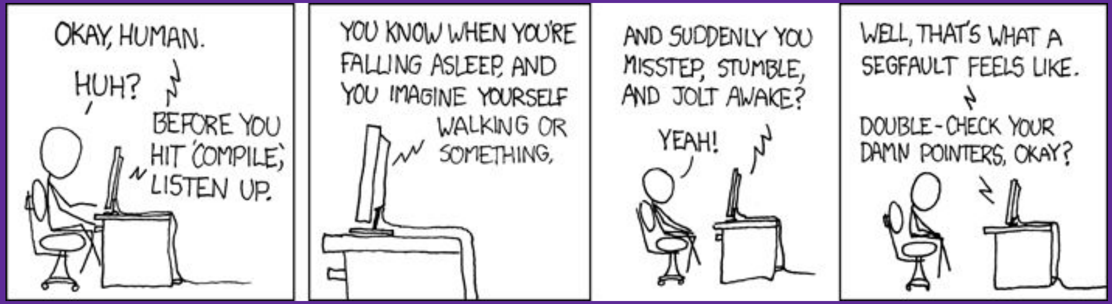
# CSE 333 Section 5

C++ Classes and Dynamic Memory



# Logistics

- Exercise 7:
  - Due **next Wednesday @ 10pm (11/1)**
- Homework 2:
  - Due **Monday @ 10pm (10/30)**
- Midterm:
  - On **Friday (TOMORROW) @ 11:30am (10/27)**
  - See Exams page on the website

# Review: Member vs. Non-Member Functions

- A **member function** is a part of the class and can be invoked on the objects of the class
- A **non-member function** is a normal function that happens to use the class
  - Often included in the module that defines the class
- Some functionality *must* be defined one way or the other, but a lot can be defined either way, so let’s examine the differences…

# Exercise 1

# Exercise 1: Member vs Non-Member Comparison

| | Member | Non-member |
|---|---|---|
| **Access to Private Members:** | Always | • Through getters and setters<br>• Through `friend` keyword (do not use unless needed) |
| **Function call (Func):** | `obj1.Func(obj2)` | `Func(obj1, obj2)` |
| **Operator call (*):** | `obj1 * obj2` | `obj1 * obj2` |
| **When preferred:** | • Functions that *mutate* the object<br>• "Core" class functionality | • *Non-mutating* functions<br>• Commutative functions<br>• When the class must be on the right-hand side |

# The "Big 4" of Classes (Review)

```
class Bar {
 public:
  Bar();                                           // 0-arg ctor
  Bar(int num);                                    // 1-arg ctor
  Bar(const Bar& other);                           // cctor
  Bar& operator=(const Bar& other);                // op=
  ~Bar();                                          // dtor
   ...
};
```

**Constructors (ctor):** Construct a new object (parameters must differ).

**Copy Constructor (cctor):** Constructs a new object based on another instance. Creates copies for pass-by-value (*i.e.*, non-references).

**Assignment Operator (op=):** Updates existing object based on another instance.

**Destructor (dtor):** Cleans up the resources of an object when it falls out of scope or is deleted.

# Construction and Destruction Details

## Construction:

1. Construct/initialize data members in order of declaration within the class.
    - If data member appears in the **initialization list**, apply the specified initialization, otherwise, default initialize.
2. Execute the constructor body.

## Destruction:

When multiple objects fall out of scope simultaneously, they are destructed in the *reverse* order of construction.

1. Execute the destructor body.
2. Destruct data members in the *reverse* order of declaration within the class.

# Exercise 2

# Exercise 2: Foo Bar Ordering

**Given these class declarations, order the execution of the program (on the next slide)**

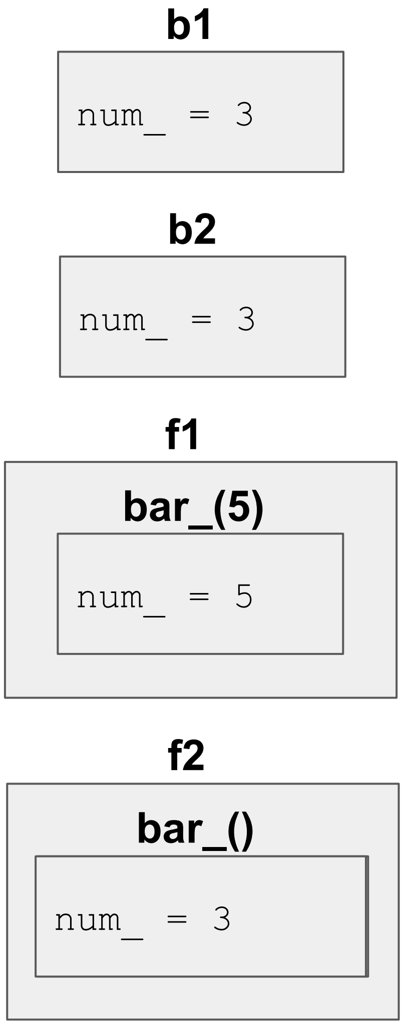
```
class Bar {
 public:
  Bar() : num_(0) { }                           // 0-arg ctor
  Bar(int num) : num_(num) { }                  // 1-arg ctor
  Bar(const Bar& other) : num_(other.num_) { }  // cctor
  ~Bar() { }                                    // dtor
  Bar& operator=(const Bar& other) = default;   // 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_;
};
```

# Exercise 2: Foo Bar Ordering

```
int main() {
  Bar b1(3);
  Bar b2 = b1;
  Foo f1;
  Foo f2(b2);
  return EXIT_SUCCESS;
}
```

**Method Invocation Order:**

1. Bar 1-arg ctor (b1)
2. Bar cctor (b2)
3. Foo 0-arg ctor (f1)
4. ↳ Bar 1-arg ctor
5. Foo 1-arg ctor (f2)
6. ↳ Bar 0-arg ctor
7. ↳ Bar op=
8. Foo dtor (f2)
9. ↳ Bar dtor
10. Foo dtor (f1)
11. ↳ Bar dtor
12. Bar dtor (b2)
13. Bar dtor (b1)



# 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(const 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

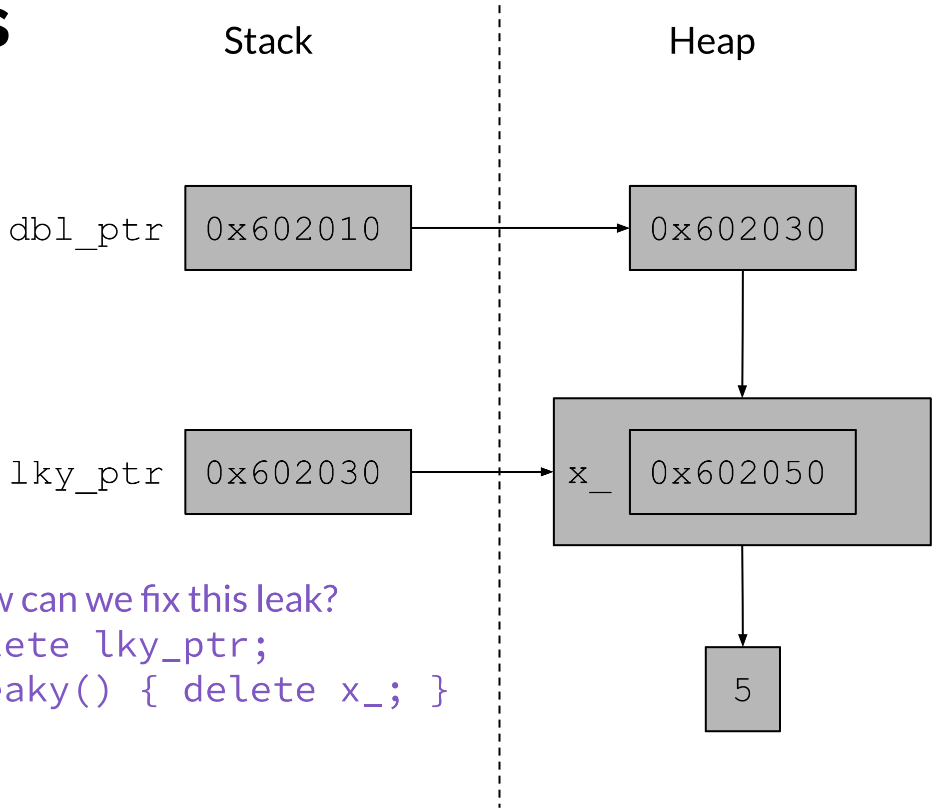
# Exercise 3: Memory Leaks

Stack | Heap

```
class Leaky {
 public:
  Leaky() { x_ = new int(5); }
 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;
  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** dbl_ptr = new Leaky*;
  Leaky* lky_ptr = new Leaky();
  *dbl_ptr = lky_ptr;
  delete dbl_ptr;
  return EXIT_SUCCESS;
}
```




How can we fix this leak?

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

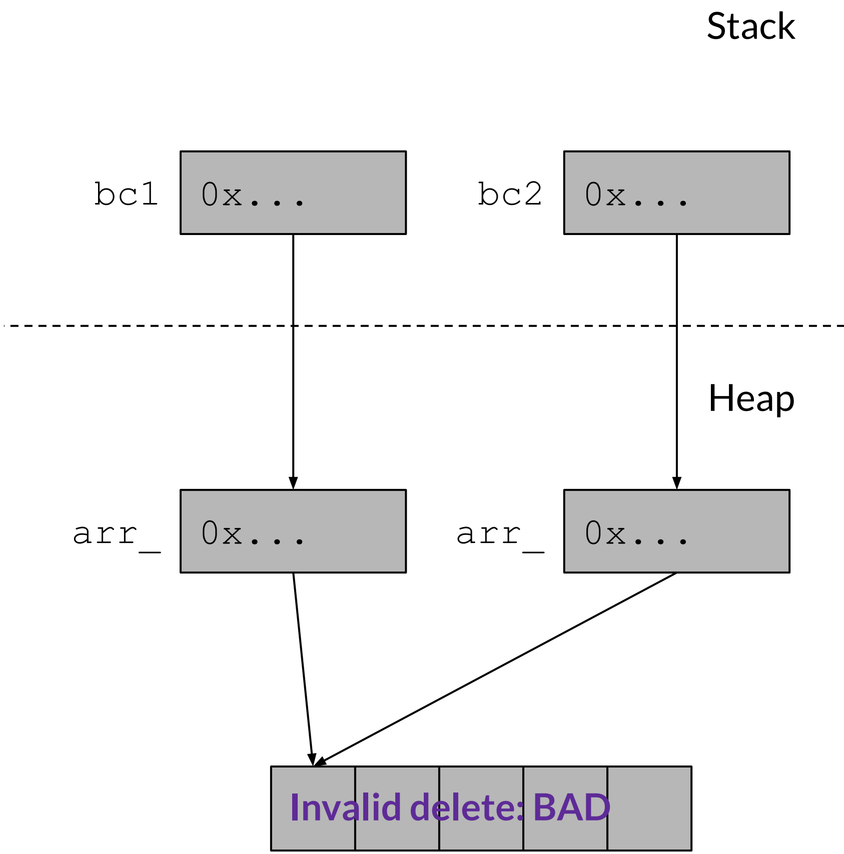
# An Acronym to Know: RAII

- Stands for "Resource Acquisition Is Initialization"
- Any resources you acquire (locks, files, heap memory, etc.) should happen in a constructor (i.e., during initialization)
- Then freeing those resources should happen in the destructor (and handled properly in cctor, assignment operator, etc.)
- Prevents forgetting to call **`free`**/**`delete`**, the dtor is called automatically for you when the object managing the resource goes out of scope.
- For more: https://en.cppreference.com/w/cpp/language/raii

# Exercise 4

# 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!
}
```



# The “Rule of Three”

- If your class needs its own destructor, assignment operator, or copy constructor, it almost certainly needs all three!
- **`BadCopy`** is a good example why, we need a destructor to **`delete arr`**, and so we needed a copy constructor too because otherwise we end up with a double **`delete`**
- **`BadCopy`** also needs its own assignment operator for the same reason, even with a fixed copy constructor, **`b1 = b2;`** would still break!
- For more info/examples, see https://en.cppreference.com/w/cpp/language/rule_of_three

# 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;
}
```

# 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 `struct` members 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 Functions

```
class Foo {
 public:
  // ctor, cctor, dtor...
  // Member function
  void MemberFunction();

  // Member operator overload
  Foo& operator*=(const Foo& rhs);
}
```

foo.h

```
void Foo::MemberFunction() {
  /* implementation */
}
Foo& Foo::operator*=(const Foo& rhs) {
  /* ... */
}
```

foo.cc

```
Foo obj1;

obj1.MemberFunction(); // call a member function

Foo obj2;

obj1 *= obj2; // call the member operator overload function
```

# Review: Non-Member Functions

```
class Foo {
 public:
  // ctor, cctor, dtor...
}

void NonmemberFunction(const Foo& f);

Foo operator*(const Foo& f1, const
    Foo& f2);
```

foo.h

```
void NonMemberFunction() {
  /* implementation */
}
Foo operator*(const Foo& f1,
    const Foo& f2) {
  /* ... */
}
```

foo.cc

```
Foo obj1;

NonMemberFunction(obj1); // invoke a nonmember function

Foo obj2;

Foo obj3 = obj1 * obj2; // invoke the nonmember operator function
```

# Review: Member vs Non-Member

| Member | Non-member |
| --- | --- |
| • Used when modifying the object (reassigning and accessing data members)<br>• “Core” class functionality | • Used for <u>non-modifying</u> and/or commutative functions.<br>• 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<br>• Only give `friend` keyword if <u>NEEDED</u><br>  ○ `friend` allows for non-member private access |
| • Function call: `obj1.Function(obj2);`<br>• Operator Overloads: `obj1 *= obj2;` | • Function call: `Func(obj1, obj2);`<br>• 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; }
};
```

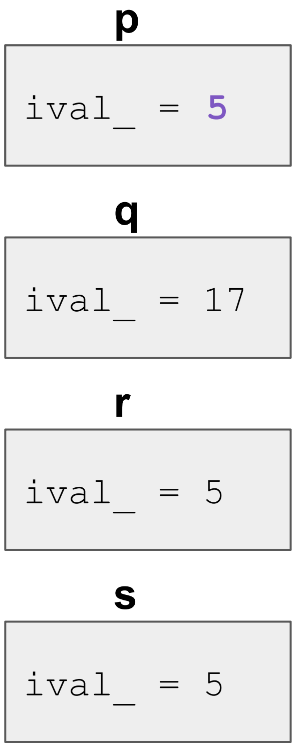
**Constructor (ctor):** Can define any number as long as they have different parameters. Constructs a new instance of the class.

**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

```
class Foo {
 public:
  Foo(int x, int y) : x_(x), y_(y) {}
  // cctor, dtor…

  // Member function
  void MemberFunction();

  // Member operator overload
  void operator*=(const Foo& rhs);
 private:
  int x_, y_;
}
```

foo.h

- Initialization lists allow a shorthand for initializing members of a class instance
- Prevents the members from being *default initialized* (which can be beneficial if the default initialization is expensive)

# 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?

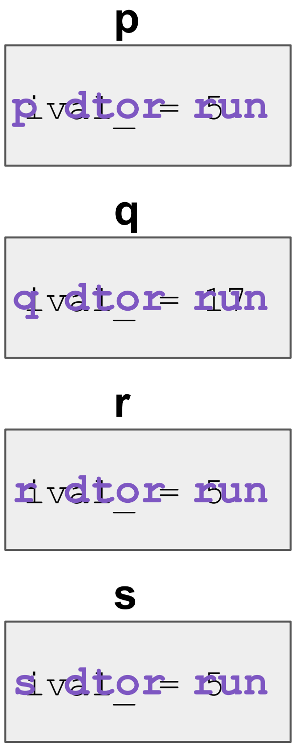
  Data members that don't appear in the initialization list are *default initialized/constructed* before ctor body is executed.

# 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 objects constructed with `new`
  - When a scope exits, local variables are destructed in reverse order of construction
- What happens when a destructor actually executes? (Hint: what happens to class 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;
}
```



# Exercise 2: 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)

p

ival_ = 17

q

ival_ = 18

r

ival_ = 5

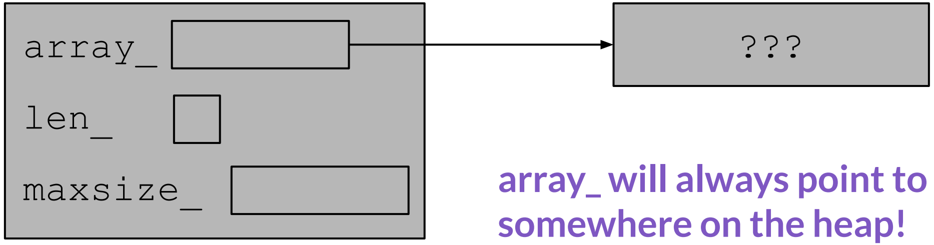
s

ival_ = 5

# Exercise 5: IntArrayList

Note: Implementation details on the worksheet

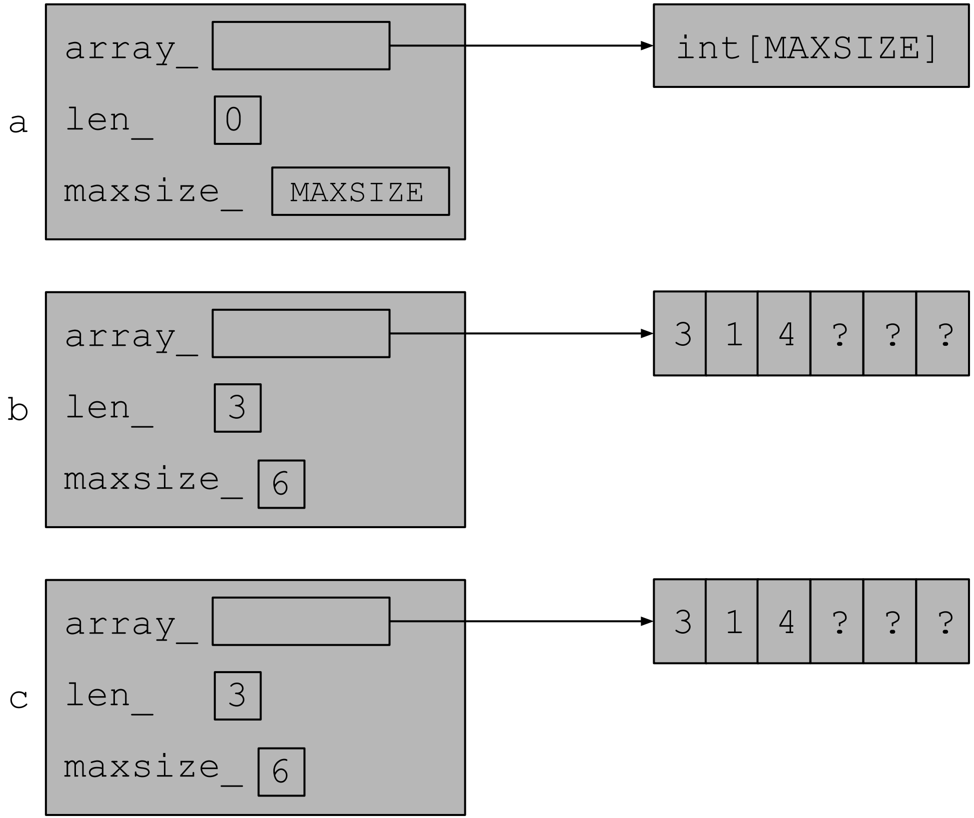
```
class IntArrayList {
  public:
    IntArrayList();
    IntArrayList(const int* const arr, size_t len);
    IntArrayList(const IntArrayList &rhs);
    // synthesized destructor
    // synthesized assignment operator

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




array_ will always point to somewhere on the heap!

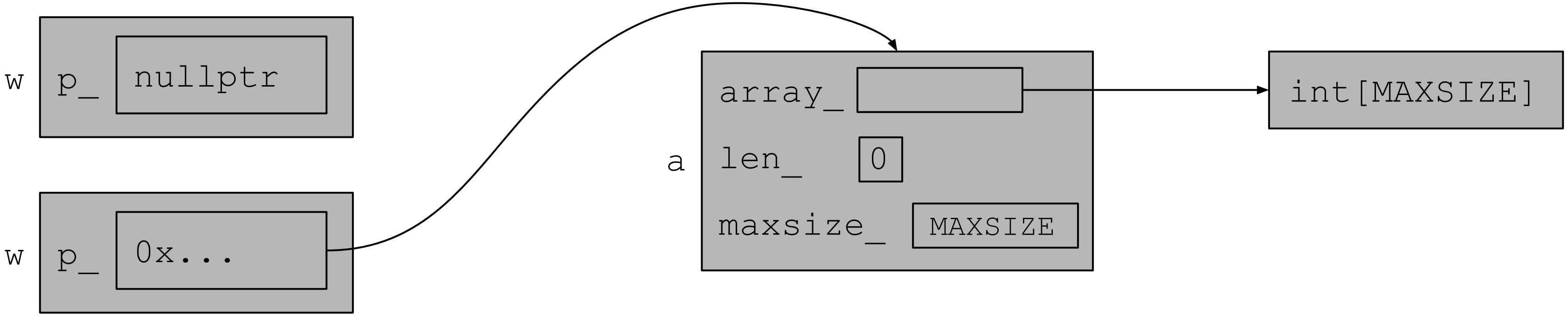
# Exercise 5: IntArrayList

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



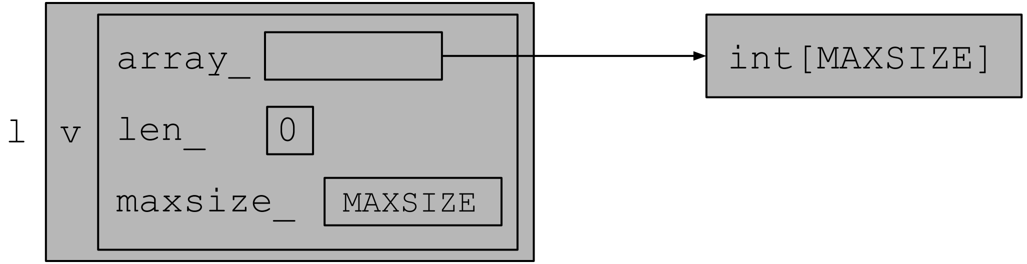
# Exercise 5: Wrap

```
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

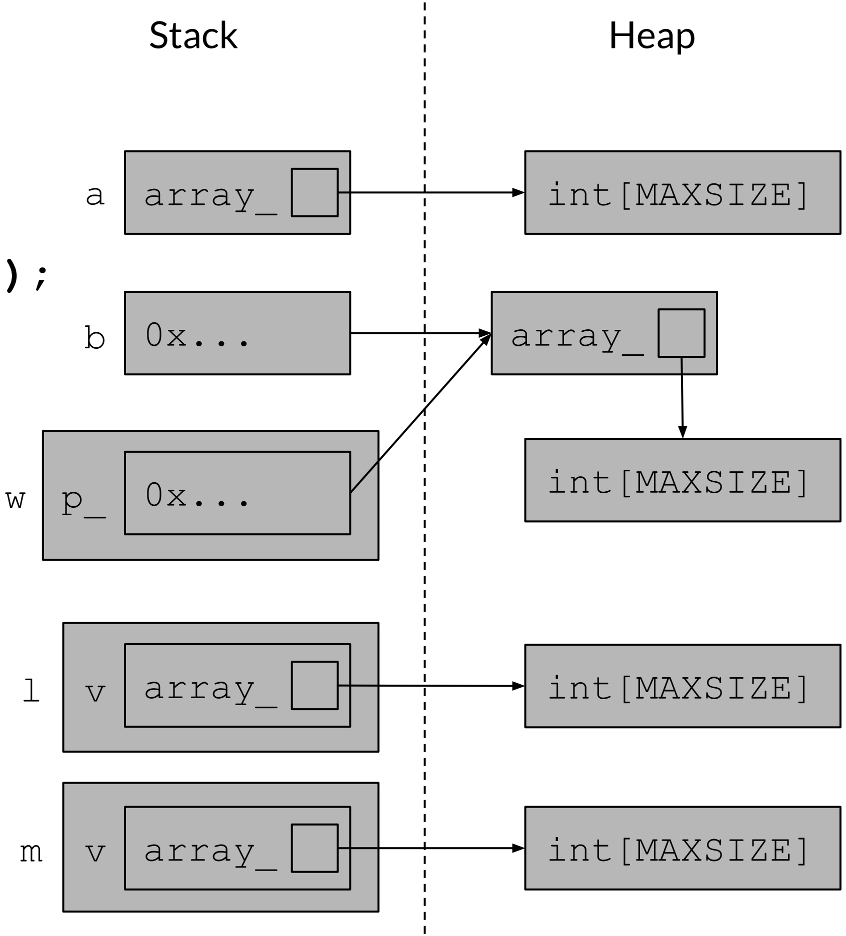
Heap

```
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

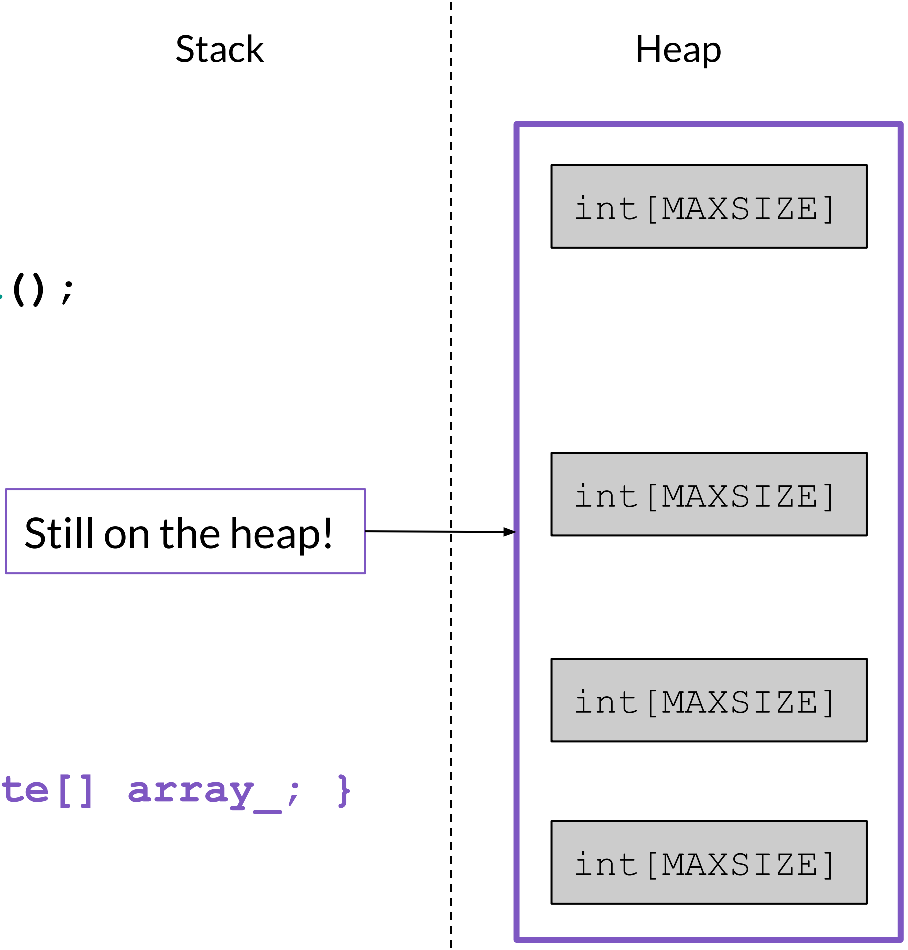
```
int main(int argc, char** argv) {
  IntArrayList a;
  IntArrayList* b = new IntArrayList();
  struct List l { a };
  struct List m { *b };
  Wrap w(b);
  delete b;
  return EXIT_SUCCESS;
}
```

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_; }
```