

C++ continued

C++ Classes and Dynamic Memory

Logistics

Thursday Oct 29 (today)

HW2 @ 11:00 pm

Monday Nov 2

Exercise 12 @ 10:00 am

Wednesday Nov 4

Exercise 12a @ 10:00 am

Section Plan - Lots of Review and Practice Today!

- C++ Classes (more)
 - constructor and destructor ordering
 - ctor, cctor, dtor, op=
- Dynamic Memory
 - new and delete
 - Drawing memory diagrams :)

C++ Classes

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 p;           // 1. default ctor  
    Int q(p);       // 2. copy ctor  
    Int r(5);       // 3. 1 arg ctor  
    Int s = r;     // 4. copy ctor (cctor)  
    p = s;         // 5. assignment operator  
}
```

p

```
ival_ = 5
```

q

```
ival_ = 17
```

r

```
ival_ = 5
```

s

```
ival_ = 5
```

Design Discussion

- 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, a default one will be synthesized for you.
 - The default copy constructor does a shallow copy of all fields.
 - The default assignment operator does a shallow copy of all fields.
 - The default 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;`

Destructors Review

When are destructors invoked? In what order are they invoked when multiple objects are getting destructed?

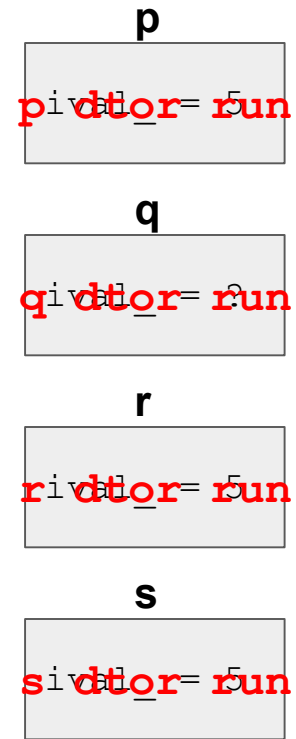
- An object's destructors is run when it falls out of scope, or when the `delete` keyword is used on heap allocated objects constructed with `new`
- Invoked in reverse order of construction

What happens when a destructor actually executes? (Hint: what happens if a dtor body doesn't destruct all 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 p;  
    Int q(p);  
    Int r(5);  
    Int s = r;  
    p = s;  
}
```



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.

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
1. 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;                default(17)  
    Int q(p);            cctor(17)  
    Int r(5);            ctor(5)  
    q.set(p.get()+1);    get(17)  
    return EXIT_SUCCESS; set(18)  
}                          dtor(5)  
                          dtor(18)  
                          dtor(17)
```

Dynamic Memory

New and Delete operators

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

```
type* ptr = new type;
```

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

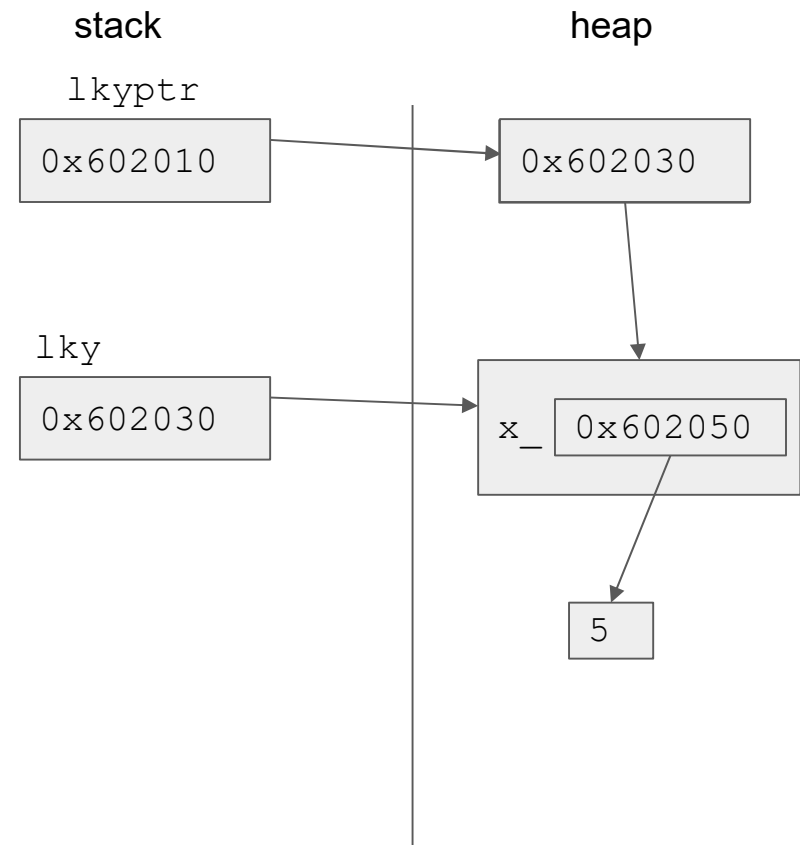
Delete: Deallocates the type from the heap, calling the destructor if it is a class type. For anything you called “new” on, you should at some point call “delete” to clean it up. Syntax:

```
delete ptr;
```

```
delete[] heap_arr;
```

Exercise 3: Memory Leaks

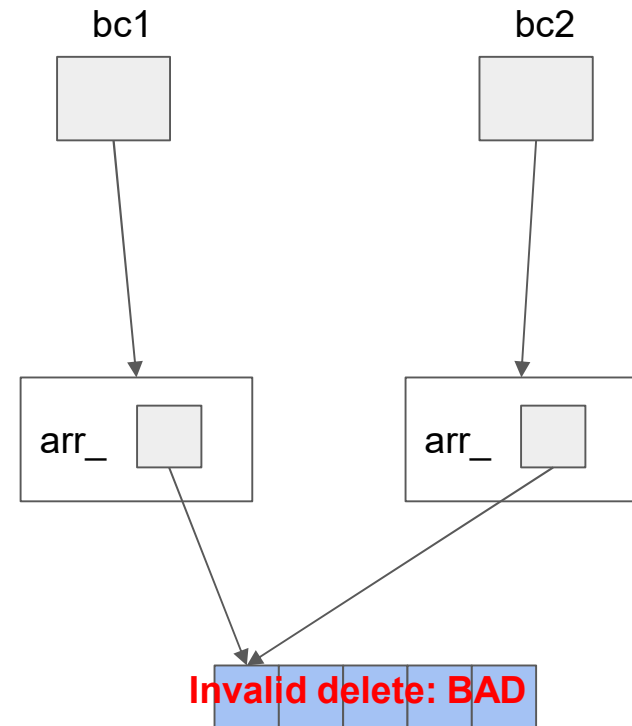
```
class Leaky {  
public:  
    Leaky() { x_ = new int(5); }  
private:  
    int *x_;  
};  
int main(int argc, char **argv) {  
➔ Leaky **lkyptr = new Leaky *;  
➔ Leaky *lky = new Leaky();  
➔ *lkyptr = lky;  
➔ delete lkyptr;  
➔ return EXIT_SUCCESS;  
}
```



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); // ctor  
➔ delete bc1;  
➔ delete bc2;  
➔ return EXIT_SUCCESS; as if!!  
}
```



Exercise 5: Classes Usage

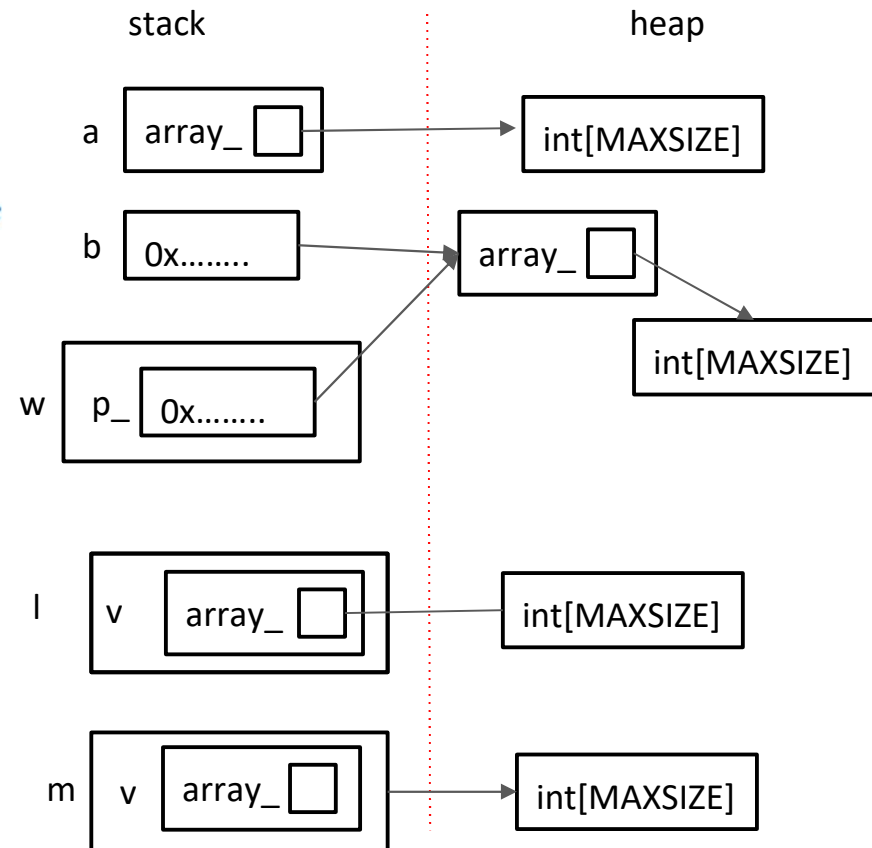
```
int main(int argc, char** argv) {  
    IntArrayList a;  
    IntArrayList* b = new IntArrayList();  
    struct List l { a };  
    struct List m { *b };  
    Wrap w(b);  
    delete b;  
    return EXIT_SUCCESS;  
}
```

Question 5

```

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;
}

```

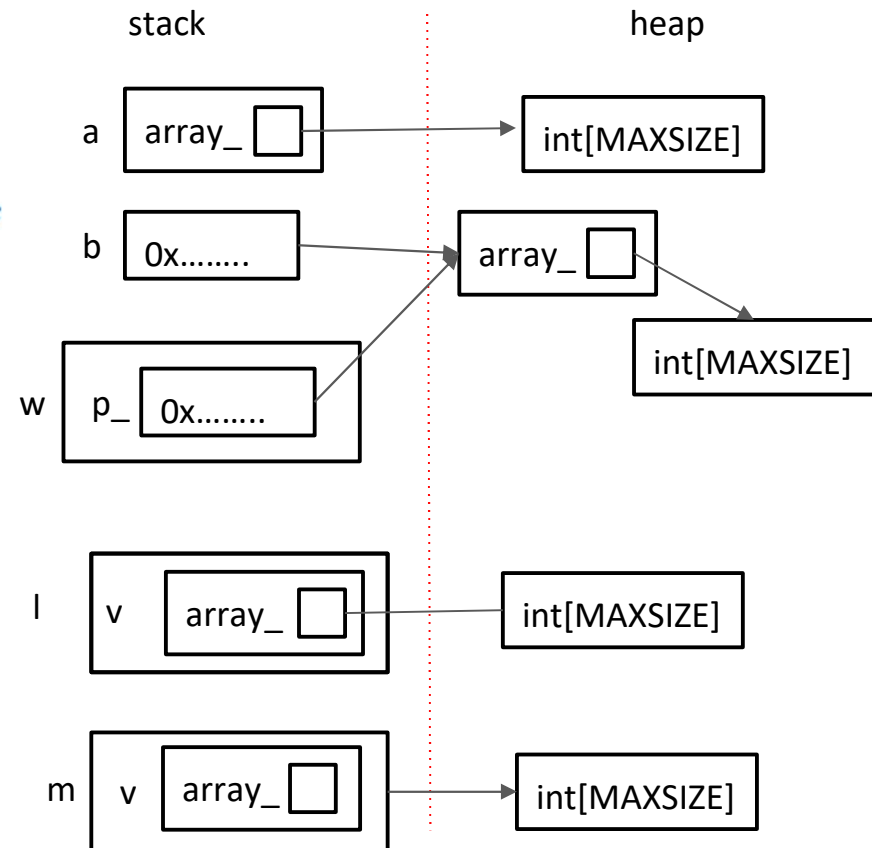


Question 5

```

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;
}

```



Question 5

```
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_; }
```

