C++ continued

C++ Classes and Dynamic Memory

Logistics

Due **TODAY**:

HW2 tonight @ 11:59 pm!

No exercise due Tomorrow!!

Due Monday:

Ex12 @ 10:00 am

Due Wednesday:

Ex12a @ 10:00 am

Questions and review

- 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 specific *foreign* functions and/or classes where this modifier is applied to.

- 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 good practice and style to use a `struct` for simple bundles of data, and a `class` for more complex abstractions.

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).

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, 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;`

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

Destructors Review

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

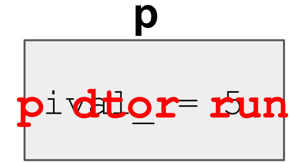
- An object's destructors is run when it falls out of scope, or when the `delete` keyword is used on heap allocated objects constructed with `new`
- Invoked in reverse order of construction

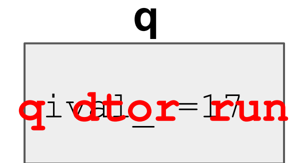
What happens when a destructor actually executes? (Hint: what happens if a dtor body doesn't destruct all its members?)

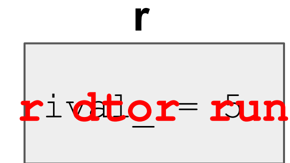
- Destructors are run in reverse order of construction: (1) run destructor body (2) destruct remaining members in reverse order of declaration

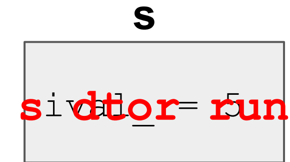
When are these destructors run?

```
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;
  Int q(p);
  Int r(5);
  q.set(p.get()+1);
  return EXIT_SUCCESS;
}
```

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

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];
```

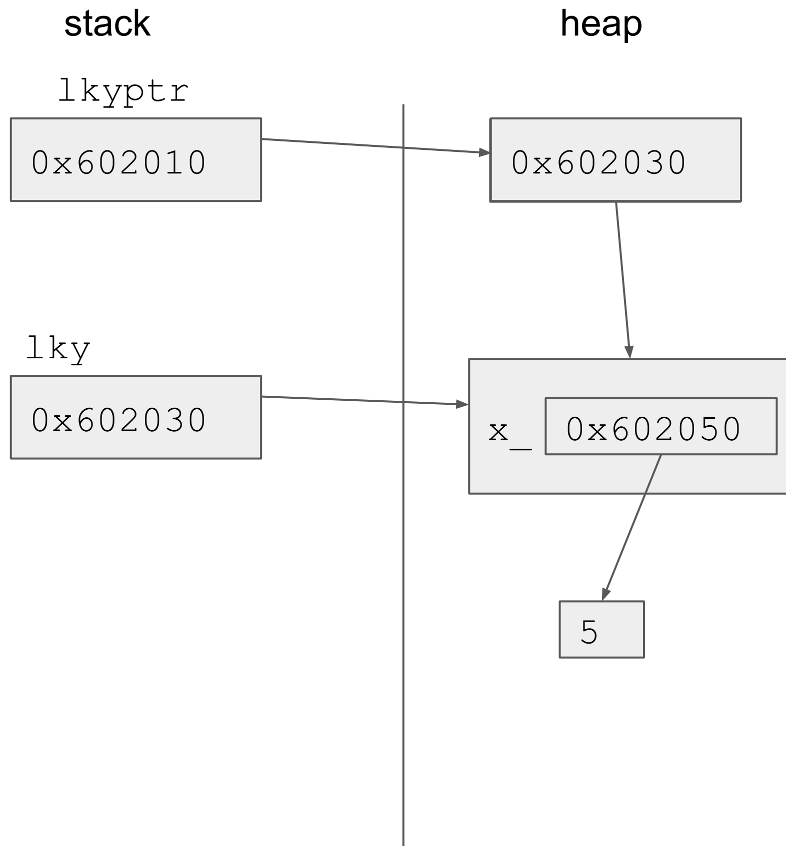
Delete: Deallocates the type from the heap, calling the destructor if it is a class type. For anything you called "`new`" on, you should at some point call "`delete`" to clean it up. Syntax:

```
delete ptr;

delete[] heap_arr;
```

Exercise 3: Memory Leaks

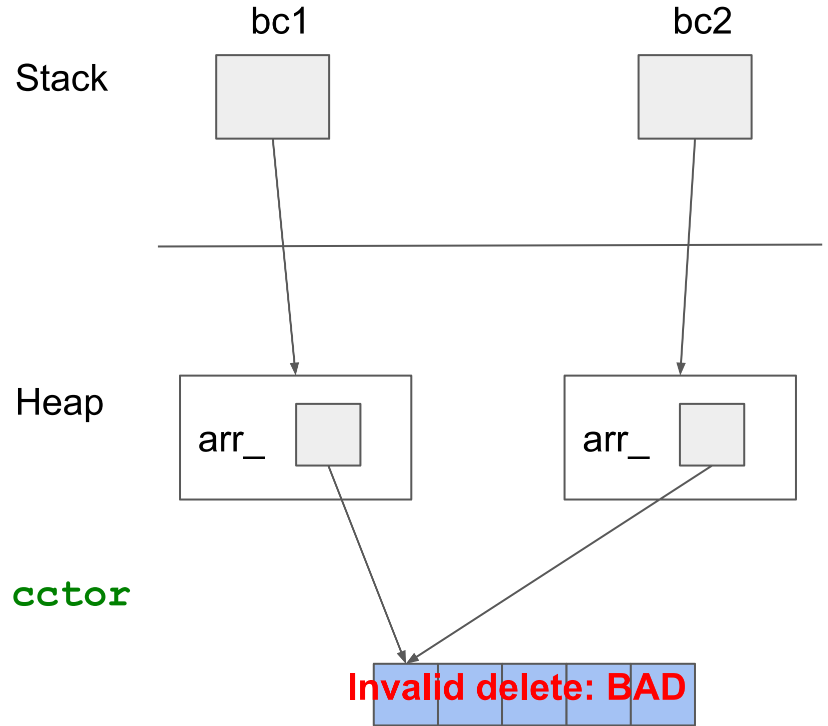
```
class Leaky {
 public:
  Leaky() { x_ = new int(5); }
 private:
  int *x_;
};
int main(int argc, char **argv) {
  Leaky **lkyptr = new Leaky *;
  Leaky *lky = new Leaky();
  *lkyptr = lky;
  delete lkyptr;
  return EXIT_SUCCESS;
}
```



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

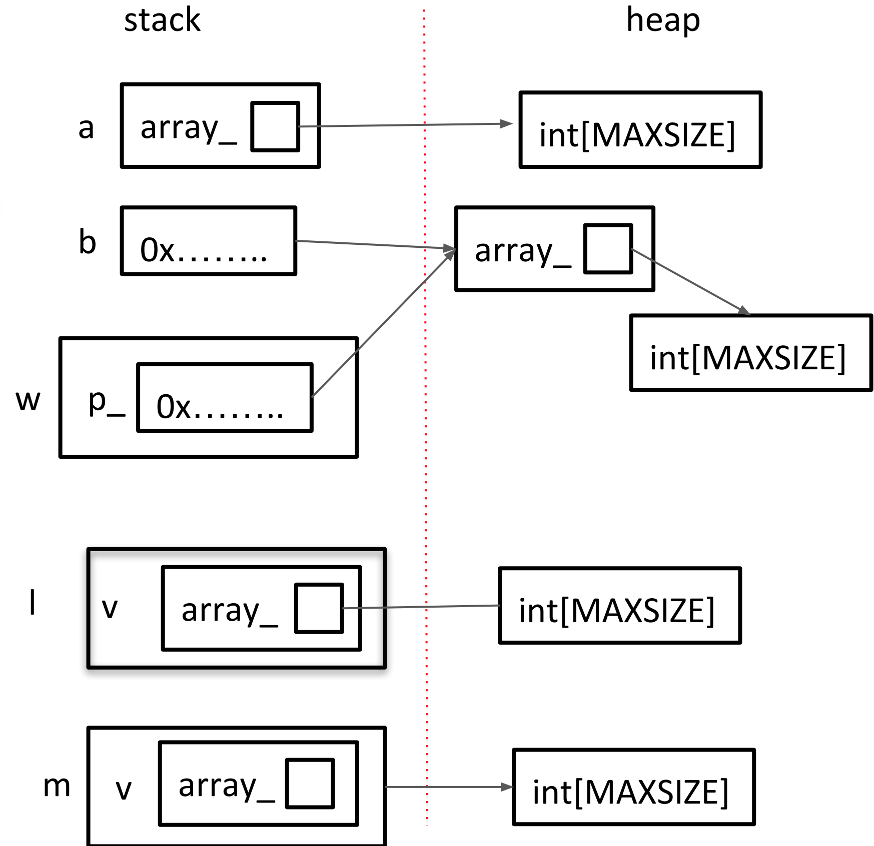


Question 5

```
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;
}
```

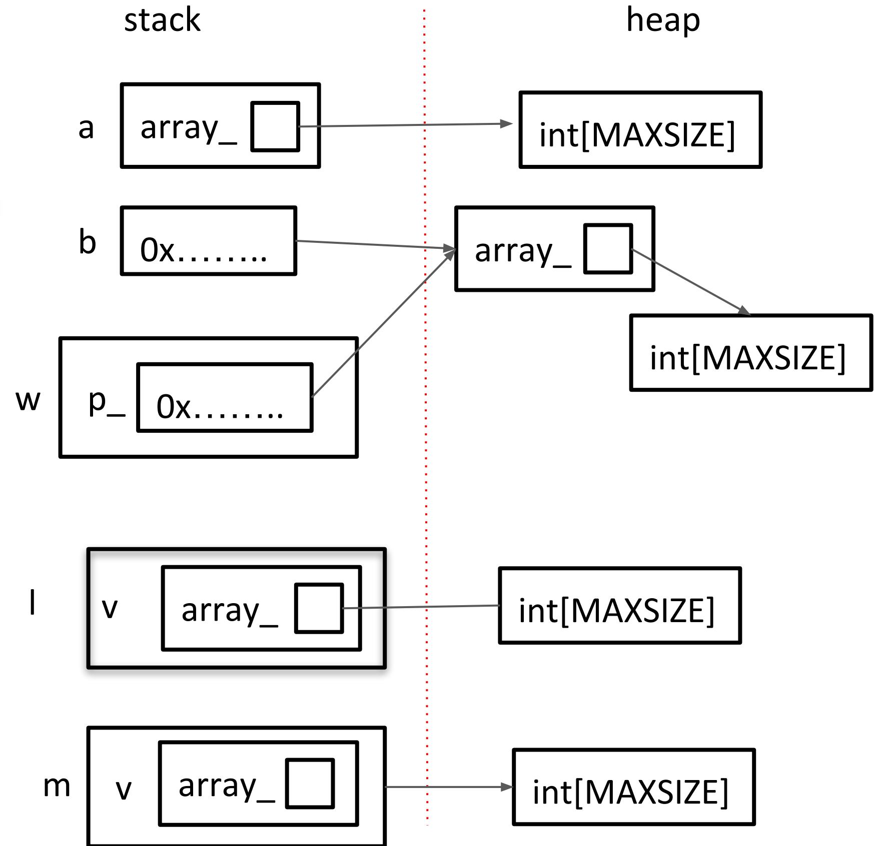
Question 5

```
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;
}
```



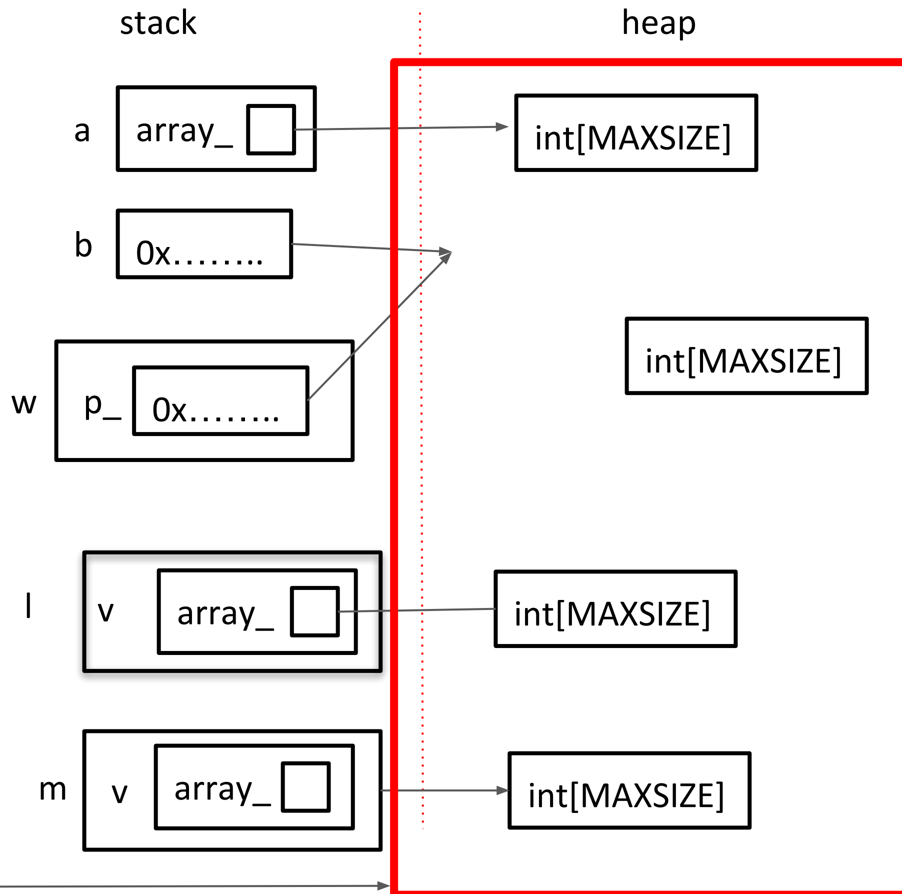
Question 5

```
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;
}
```



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_; }
```

Still on the heap!

Questions and review

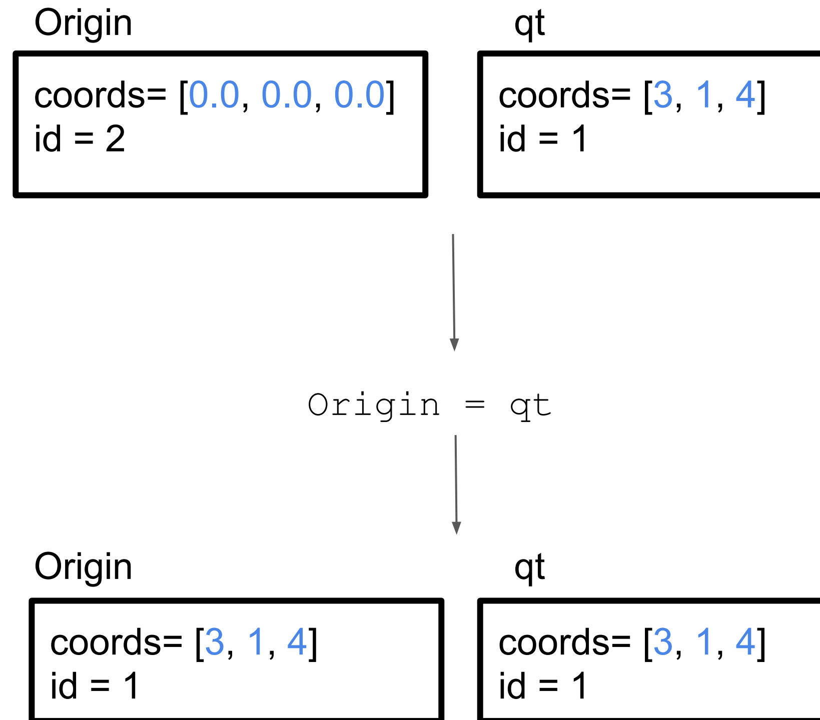
```
int main(int argc, char **argv) {
  Int p;    // default ctor
  Int q(p); // copy ctor (equivalent to: Int q = p;)
  Int r(5); // 1 arg ctor
  q.set(p.get()+1);
  // r dtor
  // q dtor
  // p dtor
}
```

- What is the destruction order?

  Destruction order is the reverse of construction order.

When we assign a `struct` variable to another, what happens when the structure contains an array?

```
struct vector {
 double coords[3];
 int id;
};
```



- Compiler automatically performs Deep Copy for array members
- Same behavior for arrays in classes

Exercise 2: Construction and Initialization

```
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;
}
```

u n g r a d e d