CSE 333
Section 5

C++ Classes and Dynamic Memory

UNIVERSITY *of* WASHINGTON

Logistics

- Homework 2
    - Due **TONIGHT! (10/27) @ 11:00pm**
- Exercise 11
    - Out Friday
    - Due **Monday (10/31) @ 10:00am**

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 (applied to individual functions/classes, not a regular access modifier like public:/private:/protected:)

- 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`

Best Practices: Member/Non-Member Functions

Member	Non-member
● Best used when you need to modify the object (reassigning and accessing data members) ● “Core” class functionality	● Best used for **<u>non-modifying</u>** 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 by default ● Only use `friend` keyword if **<u>NEEDED</u>** ○ `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, Destructors

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).

Initialization Lists

```
MyInt(int x) { ival_ = x; } => MyInt(int x) : ival(x) { }
```

- 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

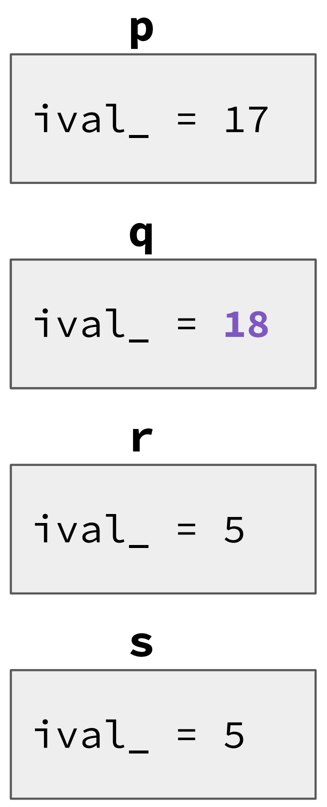
- When are destructors invoked? In what order are they invoked when multiple objects are getting destructed?
    - An object's destructor is run when it falls out of scope, or when the `delete` keyword is used on heap allocated objects constructed with `new`
    - Invoked in reverse order of construction
- 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

Exercise 1

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;`

C++ Dynamic Memory

`new` and `delete` Operators

DO NOT MIX C HEAP FUNCTIONS WITH C++ HEAP FUNCTIONS

`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 2

Exercise 2: HeapInt

Stack | Heap

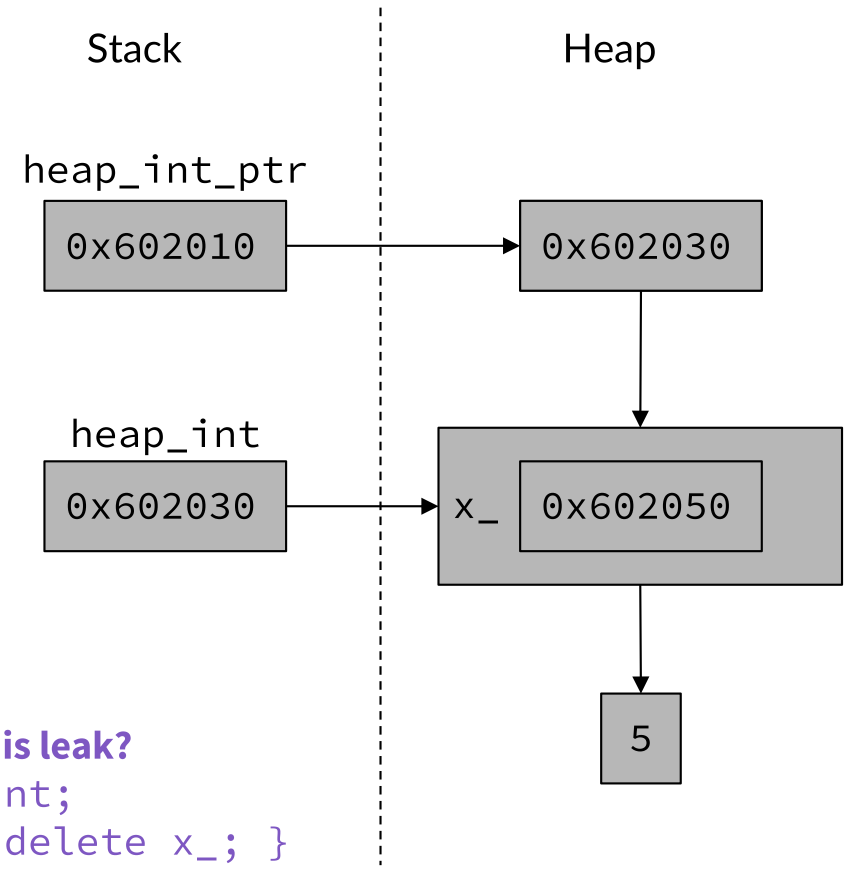
```
class HeapInt {
 public:
  HeapInt() { x_ = new int(5); }
 private:
  int* x_;
};

int main(int argc, char** argv) {
  HeapInt** heap_int_ptr = new HeapInt*;
  HeapInt* heap_int = new HeapInt();
  *heap_int_ptr = heap_int;
  delete heap_int_ptr;
  return EXIT_SUCCESS;
}
```

Exercise 2: HeapInt

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

int main(int argc, char** argv) {
  HeapInt** heap_int_ptr = new HeapInt*;
  HeapInt* heap_int = new HeapInt();
  *heap_int_ptr = heap_int;
  delete heap_int_ptr;
  return EXIT_SUCCESS;
}
```

How can we fix this leak?

```
delete heap_int;
~HeapInt() { delete x_; }
```



Exercise 3

Exercise 3: IntArr

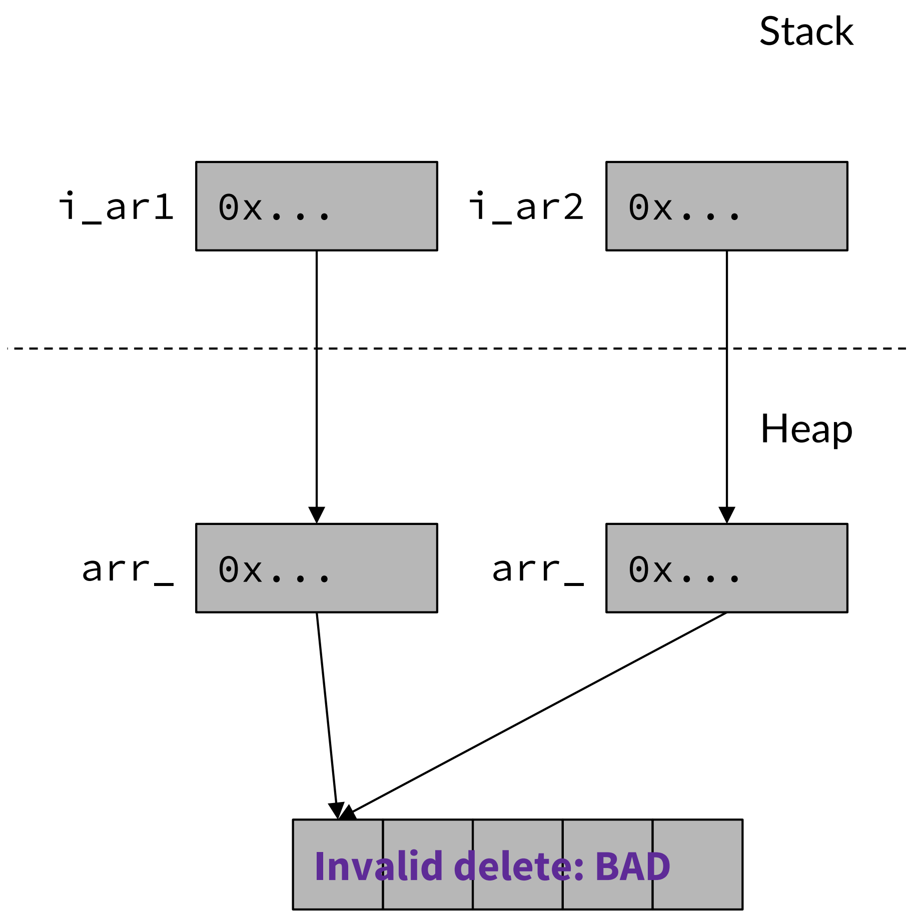
Stack | Heap

```
class IntArr {
 public:
  IntArr()  { arr_ = new int[5]; }
  ~IntArr() { delete [] arr_; }
 private:
  int* arr_;
};

int main(int argc, char** argv) {
  IntArr* i_ar1 = new IntArr;
  IntArr* i_ar2 = new IntArr(*i_ar1);
  delete i_ar1;
  delete i_ar2;
  return EXIT_SUCCESS;
}
```

Exercise 3: IntArr

```
class IntArr {
 public:
  IntArr()  { arr_ = new int[5]; }
  ~IntArr() { delete [] arr_; }
 private:
  int* arr_;
};

int main(int argc, char** argv) {
  IntArr* i_ar1 = new IntArr;
  IntArr* i_ar2 = new IntArr(*i_ar1);
  delete i_ar1;
  delete i_ar2;
  return EXIT_SUCCESS;
}
```

as if!



Stack
i_ar1
0x...
i_ar2
0x...
Heap
arr_
0x...
arr_
0x...
Invalid delete: BAD