CIT 5950 Recitation 3

Intro to C++

Logistics

- HW0 Due **TONIGHT** @ 11:59 pm
  - Don’t forget to hand in your assignment on Gradescope
  - If you need extension, please post private post on Ed
- HW1 to be released soon

Recitation

- Const & reference exercise
- Dynamic Memory Allocation: Leaky Pointer
- Object Construction & Initialization: HeapyPoint

Const and References

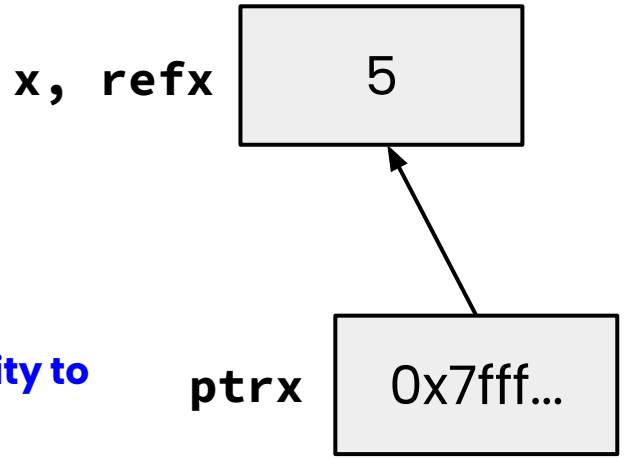
Example

- Consider the following code:

```
int x = 5;
int &refx = x;
int *ptrx = &x;
```

Note syntactic similarity to pointer declaration

Still the address-of operator!



What are some tradeoffs to using pointers vs references?

Pointers Versus References

Pointers

Can move to different data via reassignment/pointer arithmetic

Can be initialized to **`NULL`**

Useful for output parameters:
`MyClass* output`

References

References the same data for its entire lifetime - *can't reassign*

No sensible "default reference," must be an alias

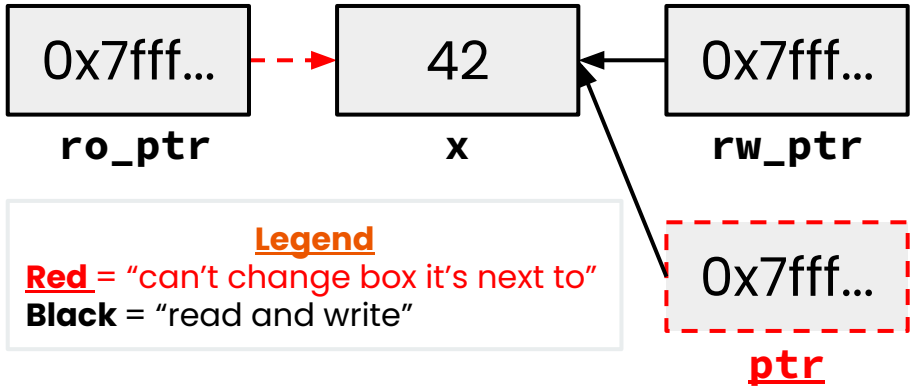
Useful for input parameters:
`const MyClass& input`

Pointers, References, and Parameters

- When would you prefer:
  - `void func(int &arg)` vs. `void func(int *arg)`
- Use references when you don't want to deal with pointer semantics
  - Allows real pass-by-reference
  - Can make intentions clearer in some cases
- Style wise, we want to use **references** for **input** parameters and **pointers** for **output** parameters, with the output parameters declared last
  - Note: A reference can't be NULL

Const

- Mark a variable with const to make a compile time check that a variable is never reassigned
- Does not change the underlying write-permissions for this variable




```
int x = 42;

// Read only
const int *ro_ptr = &x;

// Can still modify x with rw_ptr!
int *rw_ptr = &x;

// Only ever points to x
int *const ptr = &x;
```

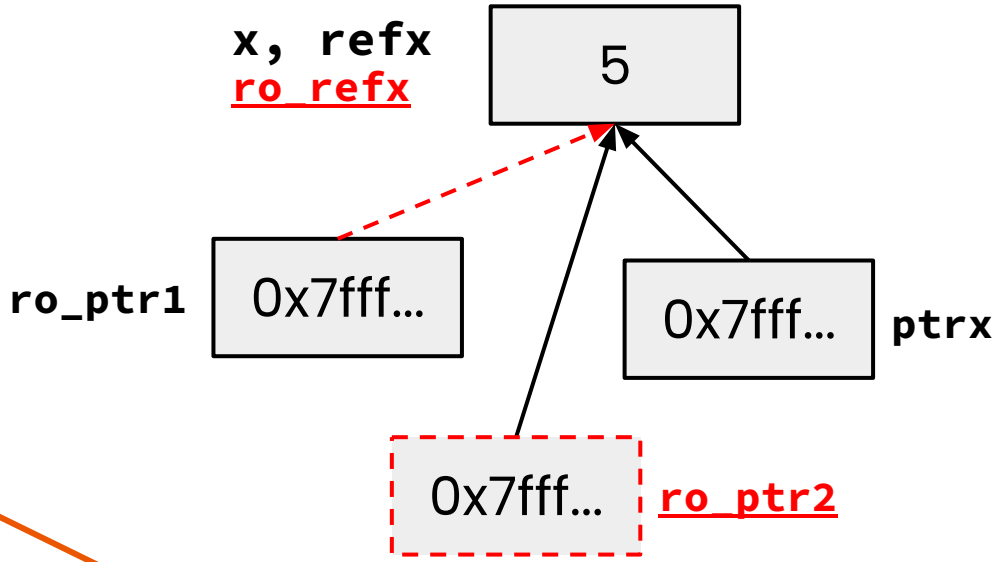
Exercise 1

```
int x = 5;
int &refx = x;
int *ptrx = &x;
const int &ro_refx = x;
const int *ro_ptr1 = &x;
int *const ro_ptr2 = &x;
```

"Const pointer to an int"

"Pointer to a const int"

Tip: Read the declaration "right-to-left"

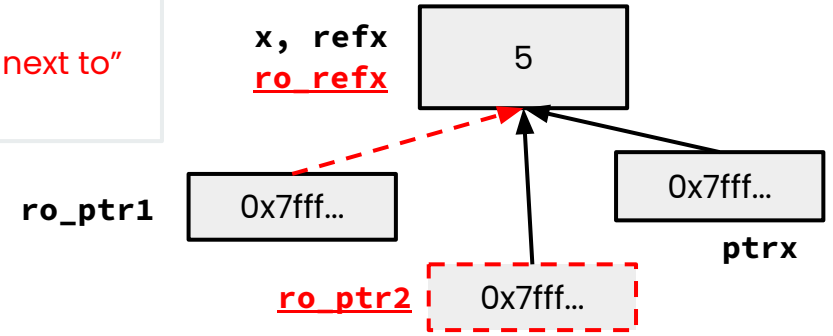


Legend

Red = "can't change box it's next to"

Black = "read and write"

Legend
Red = “can’t change box it’s next to”
Black = “read and write”



Exercise 1

```
void foo(const int &arg);
void bar(int &arg);

int x = 5;
int &refx = x;
int *ptrx = &x;
const int &ro_refx = x;
const int *ro_ptr1 = &x;
int *const ro_ptr2 = &x;
```

Which result in a compiler error?

✔ OK ✘ ERROR

- ✔ `bar(refx);`
- ✘ `bar(ro_refx);` ro_refx is const
- ✔ `foo(refx);`
- ✔ `ro_ptr1 = (int*) 0xDEADBEEF;`
- ✘ `ptrx = &ro_refx;` ro_refx is const
- ✘ `ro_ptr2 = ro_ptr2 + 2;` ro_ptr2 is const
- ✘ `*ro_ptr1 = *ro_ptr1 + 1;` (*ro_ptr1) is const

Dynamic Memory Allocation; Leaky Pointer Exercise

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 2: 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;
}
```

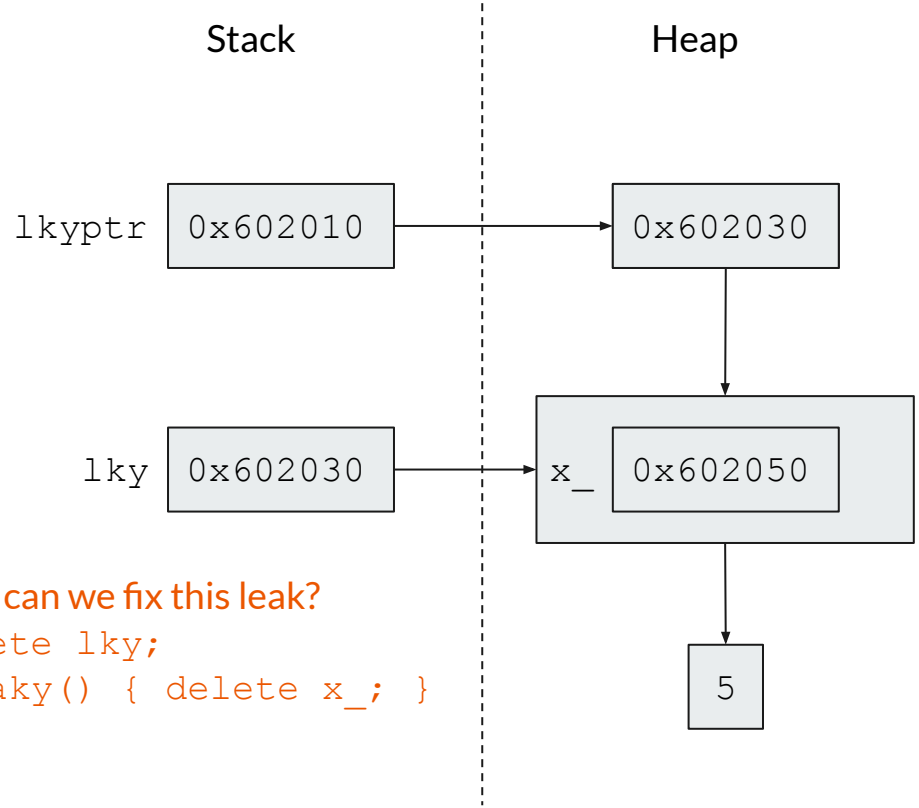
Stack

Heap

Exercise 2: 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;
}
```



How can we fix this leak?

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

Object construction; HeapyPoint Exercise

Exercise 3: HeapyPoint

Write the **class definition (.h file)** and **class member definition (.cc file)** for a class HeapyPoint that fulfills the following specifications:

Fields

- A HeapyPoint should have **three floating-point coordinates** that are all **stored on the heap**

Constructors and destructor

- A constructor that takes in **three double arguments** and initialize a HeapyPoint with the arguments as its coordinates
- A constructor that takes in **two HeapyPoints** and initialize a HeapyPoint that is the **midpoint** of the input points
- A destructor that frees all memory stored on the heap

Methods

- A method **set_coordinates()** that set the HeapyPoint's coordinates to the three given coordinates
- A method **dist_from_origin()** that returns a HeapyPoint's distance from the origin (0,0,0)
- A method **print_point()** that prints out the three coordinates of a HeapyPoint

HeapyPoint.h

```
Class HeapyPoint {

     public:
          HeapyPoint(double x, double y, double z);
          HeapyPoint(HeapyPoint& p1, HeapyPoint&  p2);
          ~HeapyPoint();
          void set_coordinates(double x, double y, double z);
          double dist_from_origin();
          void print_point();

     private:
          double * x_ptr;
          double * y_ptr;
          double * z_ptr; // pointers to coordinates on the heap

};
```

Why do we use references here?

HeapyPoint.cc - constructors & destructor

```
#include <cmath>
#include "HeapyPoint.h"
#include <iostream>


// basic constructor - three int arguments
HeapyPoint::HeapyPoint(double x, double y, double z) {
     x_ptr = new double(x);
     y_ptr = new double(y);
     z_ptr = new double(z);
}

// midpoint constructor
HeapyPoint::HeapyPoint(HeapyPoint& p1, HeapyPoint& p2) {
     x_ptr = new double ( (*p1.x_ptr + *p2.x_ptr) / 2.0 );
     y_ptr = new double ( (*p1.y_ptr + *p2.y_ptr) / 2.0 );
     z_ptr = new double ( (*p1.z_ptr + *p2.z_ptr) / 2.0 );
}
```

```
// destructor
HeapyPoint::~HeapyPoint() {
     delete x_ptr;
     delete y_ptr;
     delete z_ptr;
}
```

HeapyPoint.cc - methods

```
void HeapyPoint::set_coordinates(double x, double y, double z) {
     *x_ptr = x;
     *y_ptr = y;
     *z_ptr = z;
}

double HeapyPoint::dist_from_origin() {
     double ret = 0.0;
     ret += sqrt( pow(*x_ptr, 2) + pow(*y_ptr, 2) + pow(*z_ptr, 2)  );
     return ret;
}

void HeapyPoint::print_point() {
     std::cout << "Point: " << *x_ptr << ", " << *y_ptr << ", " <<  *z_ptr << std::endl;
}
```