CIT 5950 Recitation 1

References and Consts, Basic C++ vs C, Valgrind

Agenda

1. Logistics
2. Icebreaker
3. References/Const Review
4. C vs C++
5. Dynamic Memory Allocation Review
6. Valgrind
7. Object Construction Review

Logistics

- HW0 (simple_string and check-time) due tomorrow 1/24
  - Available until 1/28 (Add deadline) but try to finish by 24th
- Pre Semester Survey due Tuesday 1/28
- Check-in due Monday 1/27
- HW1 to be released soon

References/Const Review

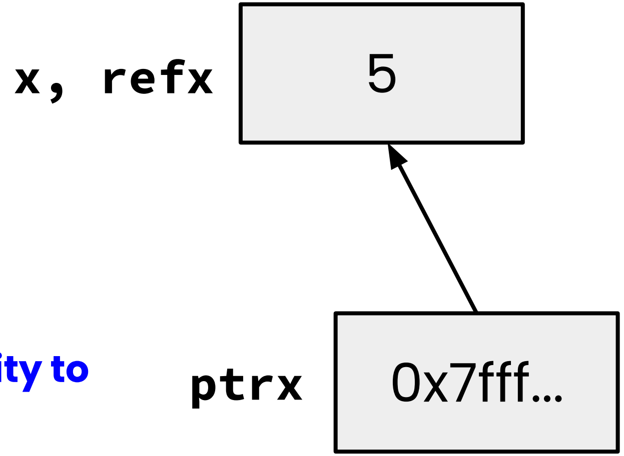
Example

- 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

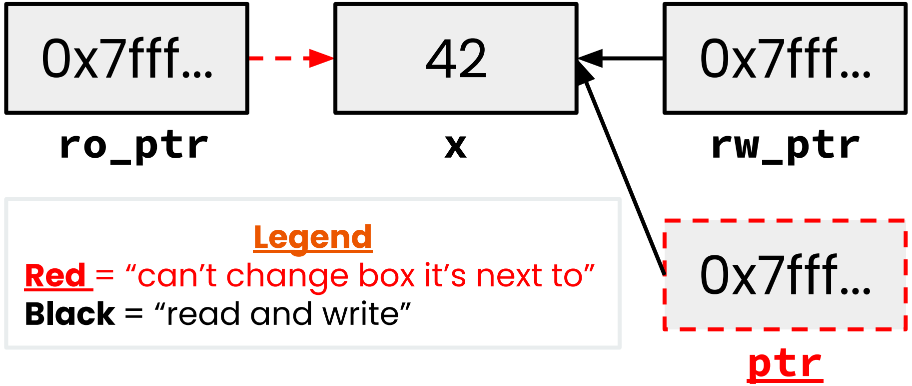
Pointers	References
Can move to different data via reassignment/pointer arithmetic	References the same data for its entire lifetime - *can't reassign*
Can be initialized to `NULL`	No sensible "default reference," must be an alias
Useful for output parameters: `MyClass* output`	Useful for input parameters: `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;

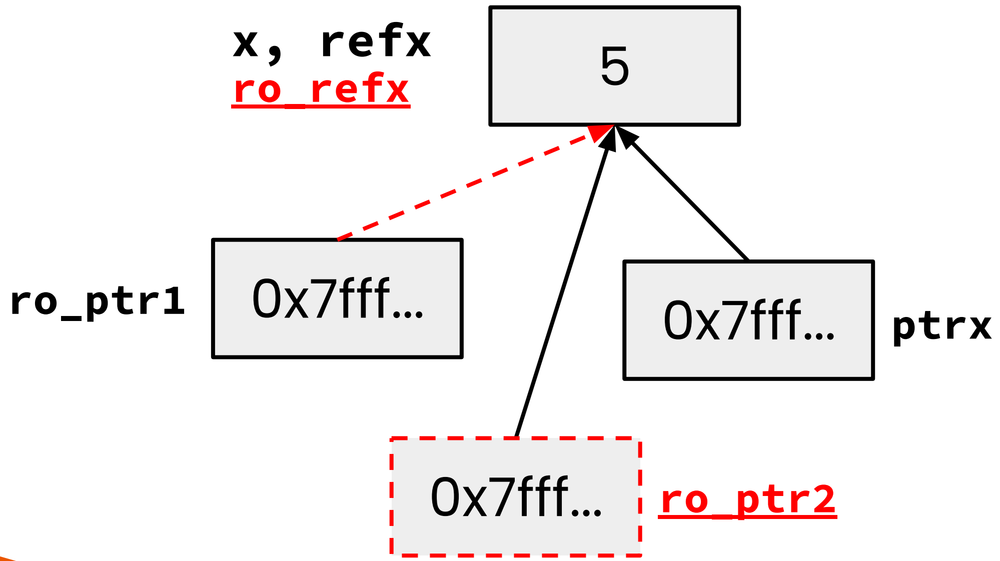
// Read only
const int *ro_ptr = &x;

// 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;
```



"Pointer to a const int"

"Const pointer to an int"

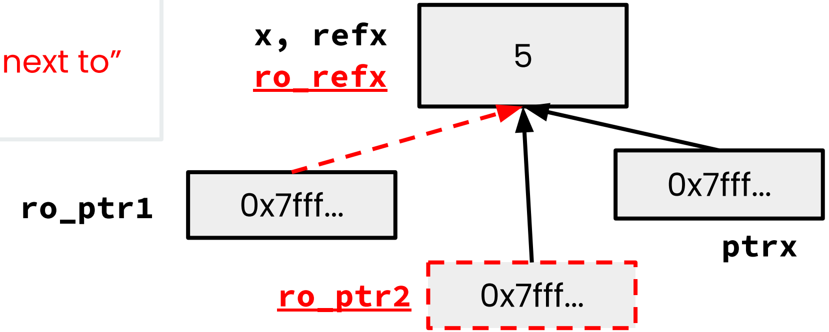
Tip: Read the declaration "right-to-left"

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

C vs C++

Memory Allocation in C++

<u>New</u>

New is used to allocate space on the heap

Requires a type and initial value

Returns pointer to beginning of allocated memory

<u>Delete</u>

Delete is used to deallocate space on the heap

Marks memory as free to use

Classes

Unlike C structs, C++ classes have methods, access modifiers, and inheritence

- Methods are functions that are members of the class
- Access modifiers affect where the methods/fields inside the class can be accessed (public vs private)
- Inheritance allows classes to derive (inherit) properties from existing classes

Constructors are used to instantiate a new object instance with the class name as the method name

- Can have multiple with different parameters

Dynamic Memory Allocation; Leaky Pointer Exercise

Why does the heap matter?

Heap is the region where dynamic memory allocation occurs.

Main Idea: Lifetime of Variables

Dynamic vs automatic allocation

```
// dynamic allocation
int* foo() {
  int* x;
  x = malloc(sizeof(int));
  *x = 595;
  return x;
}
```

```
// “Automatic” Allocation
int* foo() {
  int *x;
  int n = 595;
  x = &n;
  return x;
}
```

x would be pointed to de-allocated memory.
“n” goes away when we return

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:

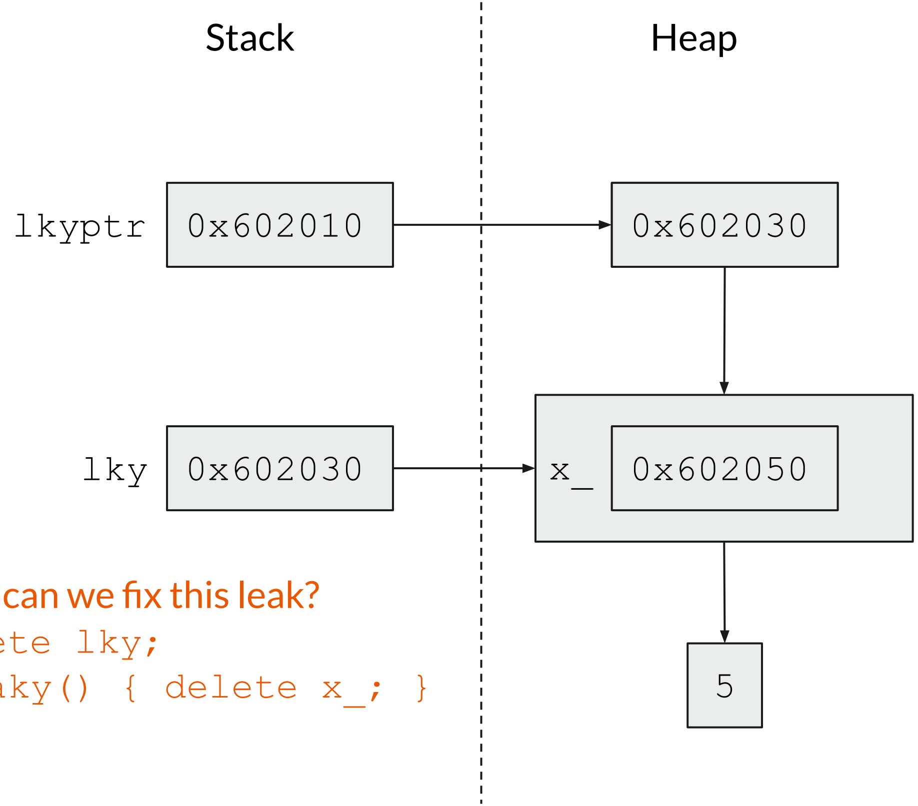
```
delete ptr;

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;
}
```



How can we fix this leak?

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

Destructors

- Automatically called when the object is out of scope or no long needed

- Deallocates memory & cleans up the class object
  - What happen if we don't call destructors - result in a memory leak

- Example syntax:
  - ```
    ~Leaky() {
        del x_;
    }
    ```
    ```

Valgrind

Valgrind for Debugging

Valgrind provides information on memory leaks and invalid memory reads/writes

- Simply run your program with valgrind before to show the output (ex: “valgrind ./compiled_program”)
  - If valgrind isn’t available install it with “apt-get install valgrind”
- When compiling, make sure the -g flag is included for detailed debugging information to be available

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:

<u>Fields</u>

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

<u>Constructors and destructor</u>

- 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

<u>Methods</u>

- 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

```
Class HeapyPoint {

    public:
        //TODO Constructor 1 three double arguments
        //TODO Constructor 2 two HeapyPoints
        //TODO Destructor
        //TODO set_coordinates()
        //TODO double dist_from_origin()
        //TODO print_point()

    private:
        //TODO Three floating-point coordinates

};
```

HeapyPoint.hpp

```
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_;
        double * y_;
        double * z_; // pointers to coordinates on the heap

};
```

Why do we use references here?

Avoid making unnecessary memory allocation for copies

(If they were passed by value, a copy of each HeapyPoint object would be created, which could be inefficient)

HeapyPoint.cpp - constructors & destructor

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


// basic constructor - three int arguments
HeapyPoint::HeapyPoint(double x, double y, double z) :
     x_(new double(x)),
     y_(new double(y)),
     z_(new double(z)) {}

// midpoint constructor
HeapyPoint::HeapyPoint(HeapyPoint& p1, HeapyPoint& p2)  :
     x_(new double((*p1.x_ + *p2.x_) / 2.0)),
     y_(new double((*p1.y_ + *p2.y_) / 2.0)),
     z_(new double((*p1.z_ + *p2.z_) / 2.0)) {}
```

```
// destructor
HeapyPoint::~HeapyPoint() {
     delete x_;
     delete y_;
     delete z_;
}
```

You can also do without initializing a list, for example

```
HeapyPoint::HeapyPoint(double x, double y,
double z) {
     x_ = new double(x);
     y_ = new double(y);
     z_ = new double(z);
}
Assignment {}:
```

Members are first default-initialized and then assigned a value. It's in some cases faster and a better practice in C++ to use initialization instead of assignment

HeapyPoint.cpp - methods

```
void HeapyPoint::set_coordinates(double x, double y, double z) {
     *x_ = x;
     *y_ = y;
     *z_ = z;
}

double HeapyPoint::dist_from_origin() {
     double ret = 0.0;
     ret += sqrt( pow(*x_, 2) + pow(*y_, 2) + pow(*z_, 2)  );
     return ret;
}

void HeapyPoint::print_point() {
     std::cout << "Point: " << *x_ << ", " << *y_ << ", " <<  *z_ << std::endl;
}
```