CIT5950 – Section 3: C++ Intro (Const, ref, memory & objects)

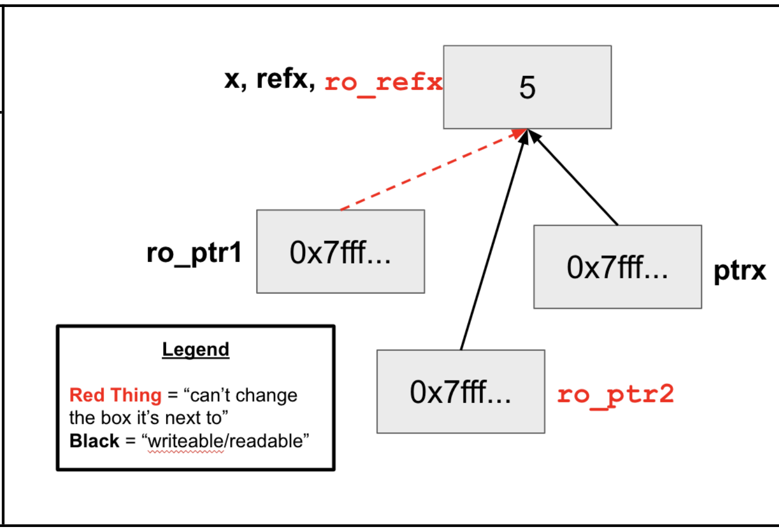
Const & References

Exercise 1: Reference & const practice

a) Draw a memory diagram for the variables declared in `main`.

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

```
int main(int argc, char **argv)
{
  int x = 5;
  int &refx = x;
  int *ptrx = &x;
  const int &ro_refx = x;
  const int *ro_ptr1 = &x;
  int *const ro_ptr2 = &x;
  // ...
}
```



b) When would you prefer `void func(int &arg);` to `void func(int *arg);`? Expand on this distinction for other types besides `int`.

- When you don't want to deal with pointer semantics, use references
- When you don't want to copy stuff over (doesn't create a copy, especially for parameters and/or return values), use references
- Style wise, we want to use **references for input parameters** and **pointers for output parameters**, with the output parameters declared last

c) What does the compiler think about the following lines of code:

```
bar(refx);                    // No issues
bar(ro_refx);                 // Compiler error - ro_refx is const
foo(refx);                    // No issues
```

d) How about this code?

```
ro_ptr1 = (int*) 0xDEADBEEF;      // No issues
ptrx = &ro_refx;                  // Compiler error - ro_refx is const
ro_ptr2 = ro_ptr2 + 2;            // Compiler error - ro_ptr2 is const
*ro_ptr1 = *ro_ptr1 + 1;          // Compiler error - (*ro_ptr1) is const
```

e) In a function `const int f(const int a);` are the `const` declarations useful to the client? How about the programmer? What about this function needs to change to make `const` matter?

The `const` return and parameter both don't affect the client at all, since they work with copies of the parameter/return value. This enforces the programmer not to modify `a` at all. If `f` used references for the parameter/return, then it would matter to both the client and the programmer.

Dynamically-Allocated Memory: New and Delete

In C++, memory can be heap-allocated using the keywords "`new`" and "`delete`". You can think of these like `malloc()` and `free()` with some key differences:

- Unlike `malloc()` and `free()`, `new` and `delete` are operators, not functions.
- The implementation of allocating heap space may vary between `malloc` and `new`.

New: Allocates the type on the heap, calling the specified constructor if it is a class type. Syntax for arrays is "`new type[num]`". Returns a pointer to the type.

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 for arrays is "`delete[] name`".

Just like baking soda and vinegar, you shouldn't mix malloc/free with new/delete.

Exercise 2: Leaky Pointer

```
#include <cstdlib>

class Leaky {
 public:
  Leaky() { x_ = new int(5); }
  ~Leaky() { delete x_; }   // Delete the allocated int
 private:
  int* x_;
};

int main(int argc, char **argv) {
  Leaky **lkyptr = new Leaky *;
  Leaky *lky = new Leaky();
  *lkyptr = lky;
  delete lkyptr;
  delete lky;  // Delete of lkyptr doesn't delete what lky points to
  return EXIT_SUCCESS;
}
```

Assuming an instance of `Leaky` takes up 8 bytes (like a C-struct with just `int* x_`), how many bytes of memory are leaked by this program? How would you fix the memory leaks?

Leaks 12 bytes of memory: 8 bytes for the allocated Leaky object `lky` points to + 4 bytes for the `int` the Leaky instance allocates in its constructor.

Deleting the `lkyptr` doesn't automatically delete what the pointer points to. Have to also delete `lky` and then create a destructor that deletes the allocated int pointer `x_`.

Exercise 3: Heapy Point

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

Class definition (in .h file):

```
Class HeapyPoint {
	public:
		HeapyPoint(double x, double y, double z);
		HeapyPoint(HeapyPoint& p1, HeapyPoint&  p2); // note the use of reference
		~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

};
```

Class member definition (in .cc file):

```
#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) { // note the use of reference
        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;
}

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;
}
```