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

Welcome to our first in-person recitation! We're glad that you're here :)

References

References create *aliases* that we can bind to existing variables. References are not separate variables and cannot be reassigned after they are initialized. In C++, you define a reference using: **`type &name = var`**. The '`&`' is similar to the '`*`' in a pointer definition in that it modifies the type and the space can come before or after it.

Const

Const makes a variable *unchangeable* after initialization, and is enforced at compile time.

```
const int x = 5;              // Can't assign to x
const int* xptr = &x;         // Can assign to xptr, but not *xptr
int *const yptr = &y;         // Can assign to *yptr, but not yptr
const int *const zptr = &z;   // Can't assign to *zptr or zptr
```

Class objects can be declared const too - a const class object can only call member functions that have been declared as const, which are not allowed to modify the object instance it is being called on.

Exercise 1: Reference & const practice

a) Draw a memory diagram for the variables declared in `main`. It might be helpful to distinguish variables that are constant in your memory diagram.

```
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`.

c) If we have functions `void foo(const int &arg);` and `void bar(int &arg);`, what does the compiler think about the following lines of code:

```
bar(refx);
bar(ro_refx);
foo(refx);
```

d) How about this code?

```
ro_ptr1 = (int*)0xDEADBEEF;
ptrx = &ro_refx;
ro_ptr2 = ro_ptr2 + 2;
*ro_ptr1 = *ro_ptr1 + 1;
```

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.

<u>Exercise 2: Leaky Pointer</u>

```
#include <cstdlib>

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;
}
```

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?

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