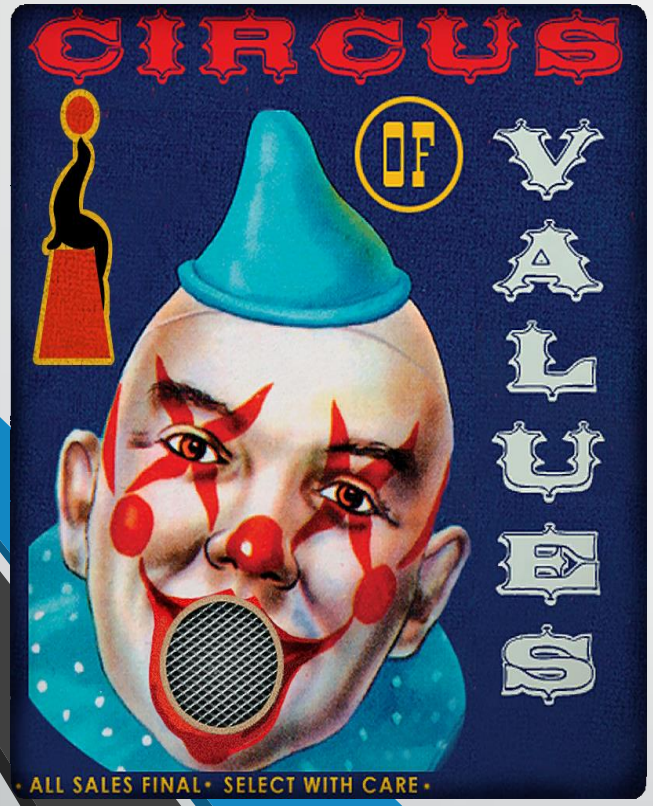


Advanced C++ Programming

Value Semantics

Basics

C++ has **Values**

- What does this mean?
  - Let's look at "01_01_basic_values.cpp"
- All types default to *value semantics*
- This means that they are *copied* when passed to or returned from functions

C++ also has **References**

- What does that mean?
  - Let's look at "01_02_basic_references.cpp"
- References *point to some specific object in memory*
- By modifying the reference, you are modifying that original object

Notes

- You can freely use any C++ type either as a value or as a reference
- This is quite different from many other mainstream languages, which might e.g. only allow value semantics for built-in types
- In C++, behaviour is uniform for all types
- This also means that the programmer needs to define what "*copying*" means for non-trivial cases

References vs. Pointers

- Same concept (reference/point to another object in memory)
- Major differences:
  - Pointers can be **nullptr**
  - References act like the object they are referencing, while pointers need to be dereferenced
  - No “reference arithmetic”

Object Lifecycle

Object Creation and Destruction

- Basics of the lifecycle
- When objects are *created*, their **Constructor** is invoked
- When objects are *destroyed*, their **Destructor** is invoked
  - Note: *The point at which the Destructor is invoked is well-defined*

- Let's look at "01_03_lifecycle_creation_destruction.cpp"

Copy Construction

- Recall: whenever a value is passed to or returned from a function, it is copied
- For objects, this happens by *Copy Construction*

- Let's look at "01_04_lifecycle_copy_construction.cpp"

Implicit Definition of Class Methods

- Why were we able to copy "Cls" in the first code example
  - We did not write a copy constructor
- The compiler implicitly provides a set of functions:
  - Default Constructor
  - Destructor
  - Copy Constructor
  - Move Constructor
  - Copy Assignment
  - Move Assignment

Rules for each on when they are implicitly generated, and when they can not be.

Why is understanding the object lifecycle important?

- Fundamental to **R**esource **A**cquisition **i**s **I**nitialization (**RAII**)
- The idea is to manage resources (such as memory, files, threads, mutexes, etc.) by leveraging the object lifecycle

- See the basics in “01_05_lifecycle_raii.cpp”

RAII Implementation Principles

- Encapsulate each resource into a class, where
  - the constructor acquires the resource and establishes all class invariants (or throws an exception if that cannot be done),
  - the destructor releases the resource and never throws exceptions
- Always use the resource via an instance of a RAII-class that either
  - has automatic storage duration or temporary lifetime itself, or
  - has lifetime that is bounded by the lifetime of an automatic or temporary object

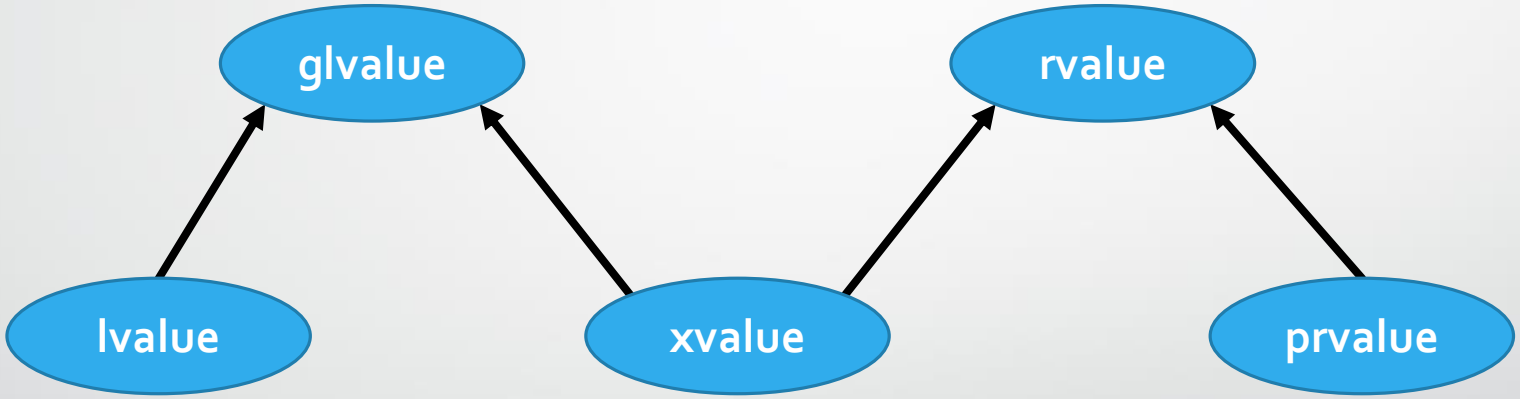
http://en.cppreference.com/w/cpp/language/raii

Advanced Values

Move Semantics

- C++ is designed to encourage *zero-overhead abstractions*
- Let's consider a simple example of a **string** class
  - Note: *you should never actually write your own "string" class*
- We'll work on "01_06_advanced_string.cpp"

Value Types



https://stackoverflow.com/a/11540204