

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”
 - References might be optimized slightly better by the compiler



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 **Resource Acquisition is Initialization (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



Advanced Values

Move Semantics

- C++ is designed to encourage *zero-overhead abstractions*
 - *Then what about all these copies happening all over the place?*
- Let's consider a simple example of a **string** class
 - Note: *you should never actually write your own "string" class*
- We'll look at "01_06_advanced_string.cpp"

Rvalue References

i) `string c(funReturningString());` ii) `string a(x);`

- How can we *safely* express the idea of reusing existing object state for i) but not for ii) ?
- `funReturningString()` generates an **rvalue**
 - This is a temporary, unnamed value we cannot assign to and cannot reuse later
- To write functions that only bind to rvalues we use *rvalue references* e.g. **`int&&`**
 - Accordingly, standard reference types like `int&` are also called *lvalue references*

Move Construction

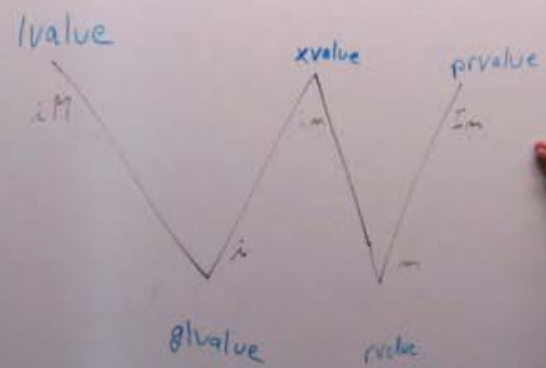
- The *move constructor* is a constructor which takes an rvalue reference type of the object
- It is called automatically **instead of the copy constructor** for the implicit construction of applicable values
- We can observe this in “01_07_advanced_string_move.cpp”

std::move

- Sometimes, we might want to move from an lvalue
 - E.g. when we know it won't be used, but the compiler does not
- `std::move` allows for that
 - Functions as a cast to an rvalue reference type
- Problem: what is the return value?
 - It has an identity, but can also be moved (that is the whole point)

Family "lvalue"

<u>iM</u>	<u>i</u>
current core WP	polymorphism
type system (reference binding)	aliasing
expression constraints	lifetime
library WP	function call
lvalue / rvalue dictionary	l-value / r-value



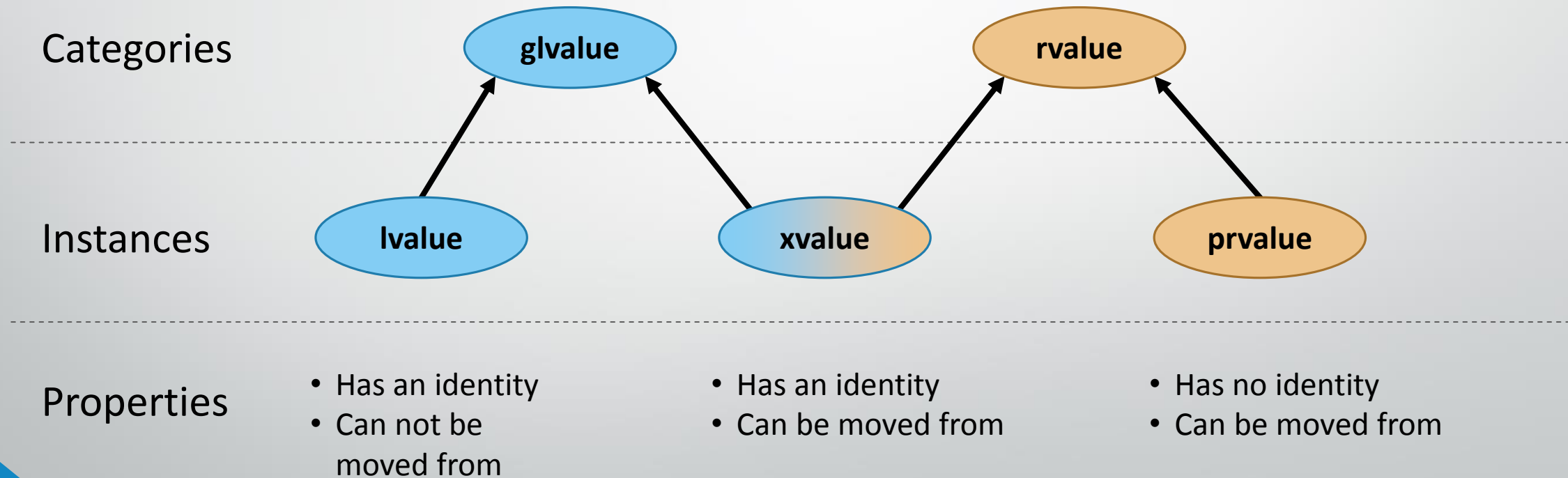
Xvalues

[...] This leaves the top middle of the W: “im”; that is, values that have identity and can be moved. We really don’t have anything that guides us to a good name for those esoteric beasts. They are important to people working with the (draft) standard text, but are unlikely to become a household name.

We didn’t find any real constraints on the naming to guide us, so we picked ‘x’ for the center, the unknown, the strange, the xpert only, or even x-rated.

“New” Value Terminology
Bjarne Stroustrup

Value Types – Overview





Conclusion & Summary

Example Real-World Use Cases

- **RAII:**
 - STL lock_guard: http://en.cppreference.com/w/cpp/thread/lock_guard
 - The simple timer I wrote for the Parallel Systems lab
(https://github.com/PeterTh/uibk_ps_parsys/blob/master/exercise02/chrono_timer.h)
- **Move semantics:**
 - std::basic_string constructors:
http://en.cppreference.com/w/cpp/string/basic_string/basic_string

Summary

- Object lifecycle:
 - Creation, destruction
 - Copy and move semantics, implicitly created functions
 - RAI
- Value categories
- References and rvalue references