C++ References

Canlin Zhang

Outline

- What reference *is*.
- What reference *isn't*.
- Why use reference when we have pointers?
- Major types of values
- Lambda capture by reference.
- Under the hood – GCC.

What is a reference?

- An **alias** for an *existing* **object**.
- Created at initialization and bound to exactly *one* object.
  - Cannot be reseated.
- Has the same type as the object it refers to.
  - This include const/volatile

What isn’t a reference?

- Not a pointer:
  - No arithmetic, not re-seatable, no nullptr.
- Not a new object [1]
- Not exactly *free* in terms of run-time cost.

[1] Implementation details often *do* store references as a pointer

Demo: Basic properties

Reference vs. Pointer

Reference

- Access like normal variable
- No reseating
- Cannot be null (theoretically)
- Stronger aliasing guarantees for compiler

Pointer

- Needs dereference
- Can be reassigned
- Can be null

Demo: Reference vs. Pointer

Clarification: Value Types vs. Object

Value Types

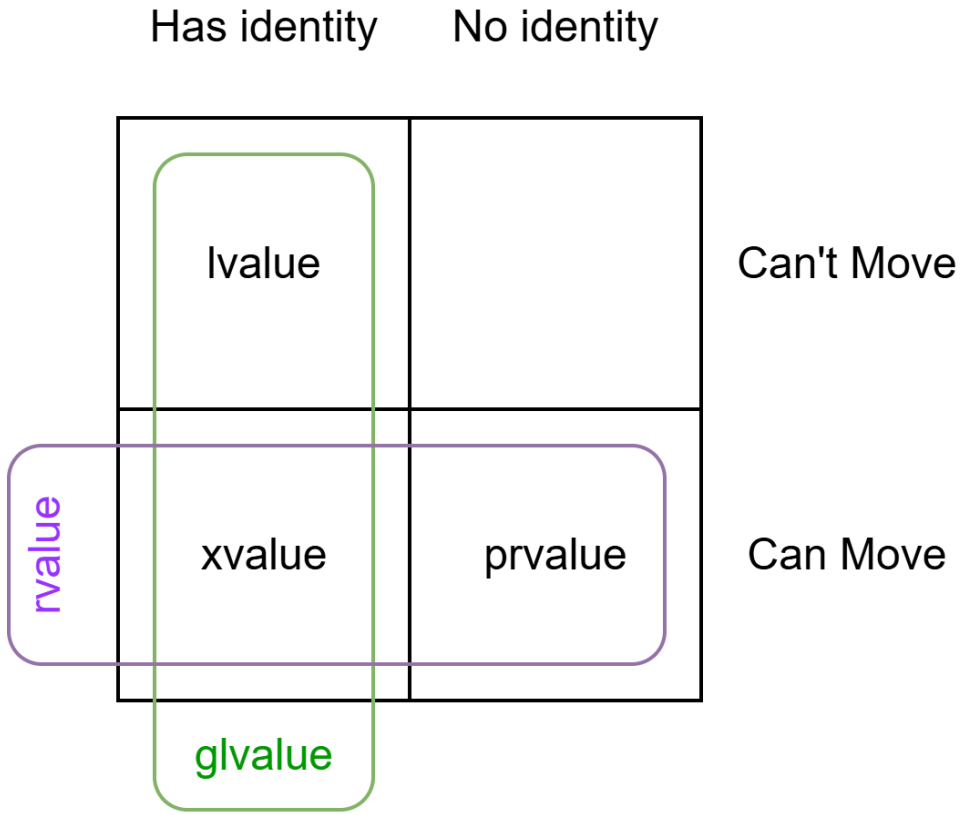
- Describes *how* the expression behaves in terms of
  - identity (whether it refers to a specific object)
  - movability (whether its resources can be "stolen" or reused).

Object

- A region of storage (in memory) that hold a value of a specific type.

There is NO lvalue/Prvalue/xvalue objects. These category only exists for expressions.

C++ Value Types: Glvalue

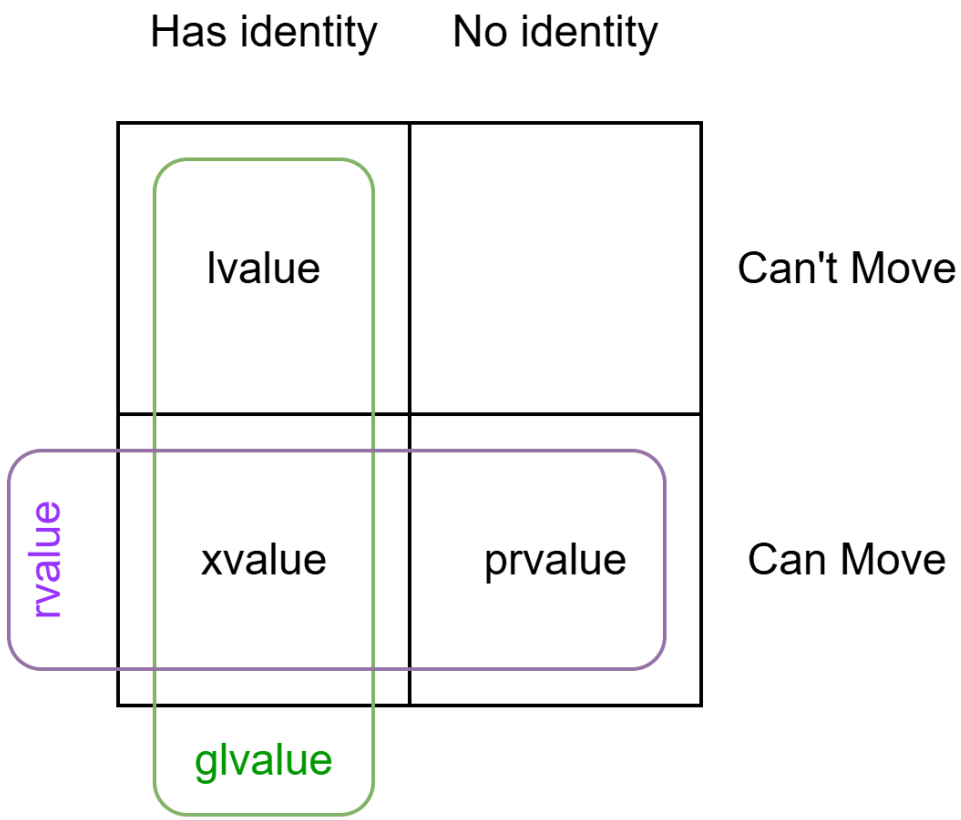


Value Type	Definition
Glvalue (generalized lvalue)	An expression whose evaluation determines the identity of an object or function. Such an expression refers to a persistent entity in memory.
Xvalue (expiring value)	A glvalue expression that refers to an object (often accessed by lvalue expression) whose resources may be reused (i.e., expire).
Lvalue	A glvalue expression that is not an xvalue expression.

Ref: https://blog.panicsoftware.com/value-categories-gl-l-x-r-prvalues/

Demo: Glvalue examples

C++ Value Types: Rvalue



Value Type	Properties
Rvalue	An expression that does not determine a persistent identity. Rvalues are typically used to initialize new objects or to indicate that resources may be moved from.
Xvalue	A glvalue expression that refers to an object (often accessed by lvalue expression) whose resources may be reused (i.e., expire). In this case, its “Rvalue-ness” comes from the fact that it’s safe to move from.
Prvalue	A rvalue expression that creates a temporary or computes a value but does not have persistent identity.

Ref: https://blog.panicsoftware.com/value-categories-gl-l-x-r-prvalues/

Demo: Rvalue examples

Putting it together: std::move()

```
// We need to bind it to a const lvalue refere
const std::string &s = std::string(1000, 'A');

// Print address of s and s's data container
std::cout << "Address of s (i.e., the prvalue)
std::cout << "Address of s's data: " << static

// We move s using an xvalue expression
auto t = std::move(s);
```

- Object (nameless):
  - “O1”: std::string objects that live inside memory (stack + heap), created upon a prvalue expression
  - “O2”: another std::string object that live inside memory, created by copy construction from “O1”.
- Expression:
  - “s”: a const lvalue expression bound to “O1”
  - “std::move”: an xvalue expression bound to O1 (identified by “s”) but marks it as expiring (resources may be moved)
  - “t”: a lvalue expression bound to “O2”, O2 is created and initialized by a copy constructor by operator “=“ based on the return type of std::move(“s”)

Lambda capture by reference

Phenomenon

1. Reference capture never extends the lifetime of the referent
2. Capturing the same loop variable by reference makes all closures see its final value.
3. [&] or [this] captures the pointer this (still reference-like risks if the object dies).

Reason

1. Lifetime: A reference capture stores an alias to someone else's object; it doesn't create one.
2. Loop indices: There's one loop variable object; by-ref captures all point to it. After the loop, that one object has the last value.
3. "this" capture: [this] stores a pointer to the current object inside the closure. If the object is destroyed before the callback runs, the pointer dangles.

Under the hood: The GCC Way

Two lines inside gcc/tree.def:

/* A reference is like a pointer except that it is coerced automatically to the value it points to. Used in C++. */

DEFTREECODE (REFERENCE_TYPE, "reference_type", tcc_type, 0)

Gist of actual implementation: Pointer with extra step.

Under the hood: The GCC Way

Planning

Inside decl.cc

1. cp_finish_decl – Start of semantic finalization for the decl.
2. check_initializer – Routes to special init paths (like for references).
3. grok_reference_init – Detects this is a reference and sets up the binding call.

Inside call.cc

4. initialize_reference – General entry point for binding references.
5. reference_binding – Implements the C++ standard binding rules.
6. convert_like – Executes each step in the conversion object.

Execution & Usage

Inside cvt.cc

6. convert_to_reference – Converts the expression to the reference type.
    1. Checks qualifiers.
    2. Calls check_for_null_reference() for constant-null.
    3. Calls build_up_reference() to do the actual pointer wrapping.
7. build_up_reference – Creates the hidden pointer value and wraps it as REFERENCE_TYPE.
8. convert_from_reference – When you use the reference, automatically inserts the dereference (INDIRECT_REF) to get the underlying object.