C++ Value Categories in Action

The Foundation Behind Overloads, Moves, and the Broader System

Mantrala Sandeep

2025

C++ Value Categories

In Action

Foundation behind overloads, moves and the broader system

Sandeep Mantrala 23/06/2025

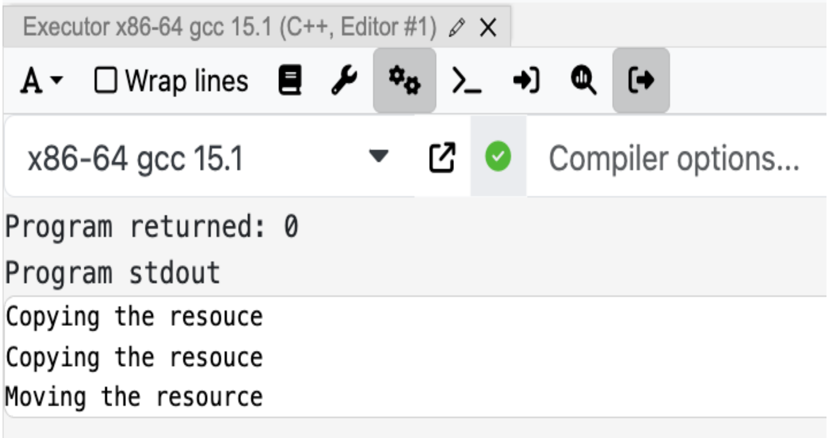
Coming Up...

- Value Categories, Reference bindings & Overload resolution
- Move semantics and Universal references
- Best practices

Quiz time!

```cpp
class Pool {
  std::vector<Resource> resources;

public:
 void addResource(const Resource& r) {
    std::cout << "Copying the resouce\n";
    resources.push_back(r);
 }

 void addResource(Resource&& r) {
    std::cout << "Moving the resource\n";
    resources.push_back(std::move(r));
 }
};
```

```cpp
//Which Overload is called?
Pool pool{};
Resource a{"Alpha"};
pool.addResource(a);
// copying the resource

Resource && b{"Beta"};
pool.addResource(b);
//Copying the resource

pool.addResource(Resource{"Gamma"});
//Moving the resource
```



https://godbolt.org/z/hazEn6hss

Why Learn About Value Categories ?

- Fundamental to most **modern C++ concepts**
- **Read Compiler errors** effectively
- Helps you **write semantically correct and performant code**

So, What Are Value Categories About?

❌ **Value Categories Are Not About...**

Objects or Variables

Class types

lifetimes

✅ **Value Categories Are About...**

</> Expressions

</> Expressions

</> Expressions

So.. What is an Expression?

*"An expression is a **sequence of operators and their operands**, that specifies a computation."*

*Expression evaluation may **produce a result**, and may **generate a side-effect.***

💡 A piece of code that evaluates to a value and might have side effects

Expression Examples

Valid Expressions

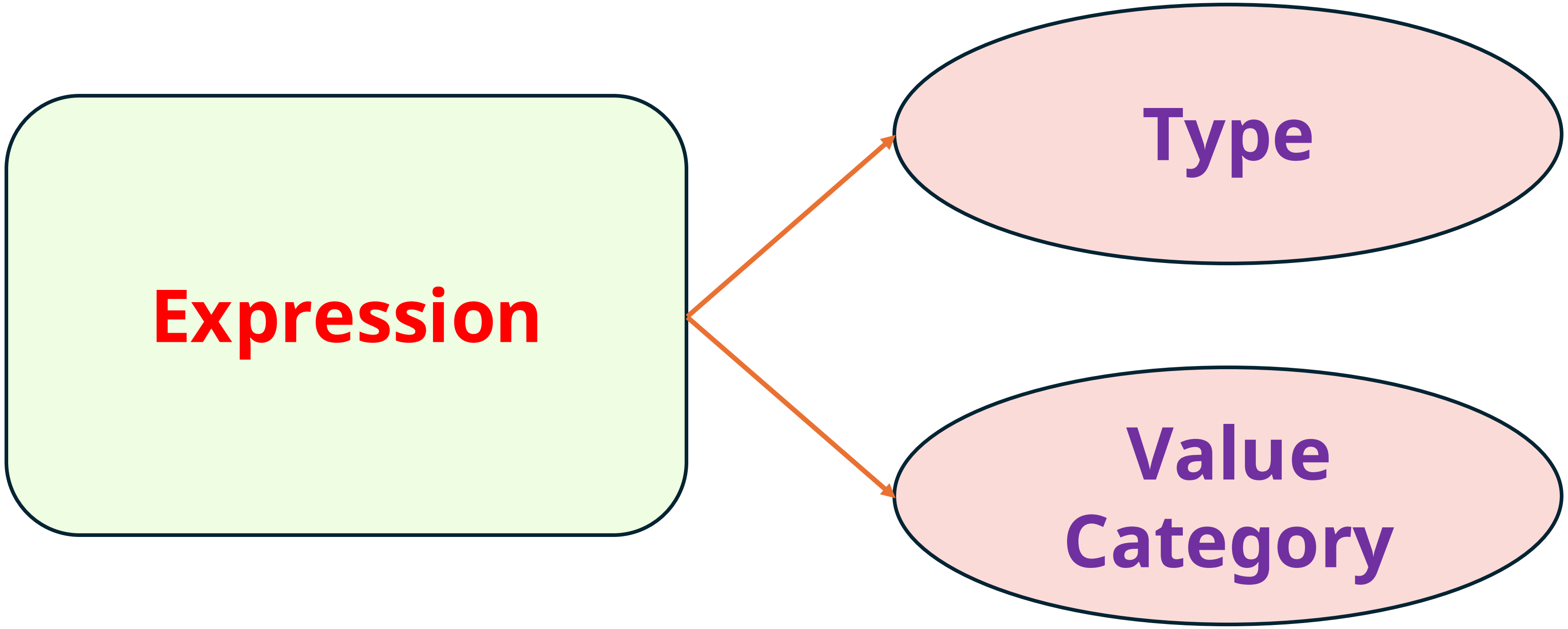
```
// Evaluating to a value
5 + 3 // evaluates to 8
std:string("hi") // Temporary string object
func() // evaluates to the return value
[](){return 42;}() // closure object

// Accessing existing objects
*ptr
arr[2]
std::move(x)

// causing side effects
x = 5 // assigns 5 to x
++x // increments x by 1
std::cout << x // prints to standard output
```

Not Expressions

```
int a; //statement
namespace ns {} //scope declaration
void fun () {} // Function definition
typedef int MyInt; // Type alias
class MyClass {...
}; // Type definition
```

Only expressions have value categories!

Every Expression Has Two Independent Properties



What Kind of value?

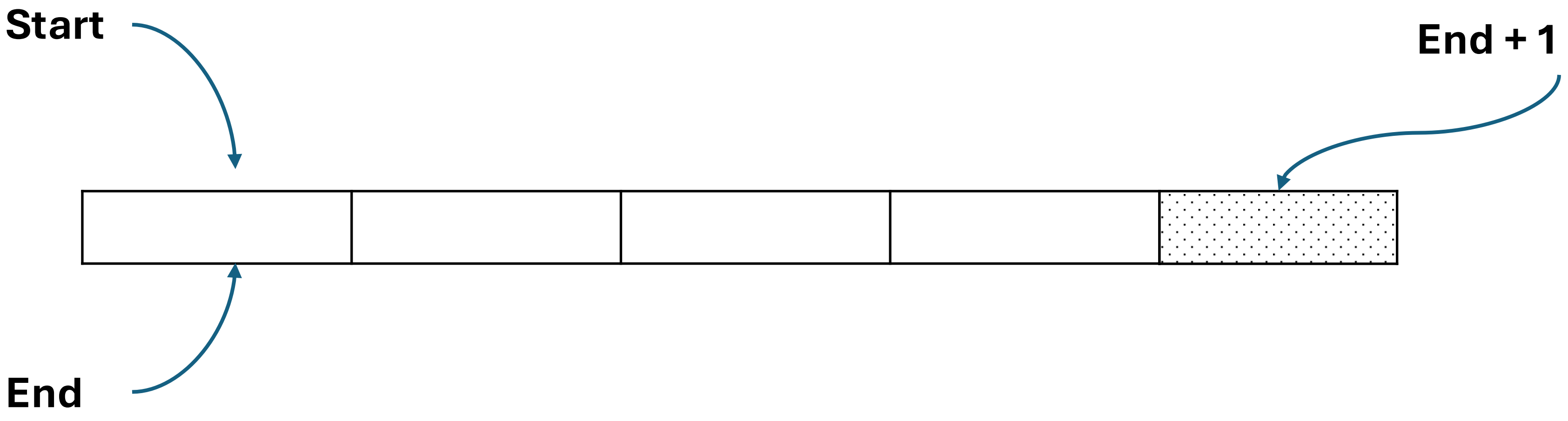
`int, const int, int &, int && etc..`

How does it behave?

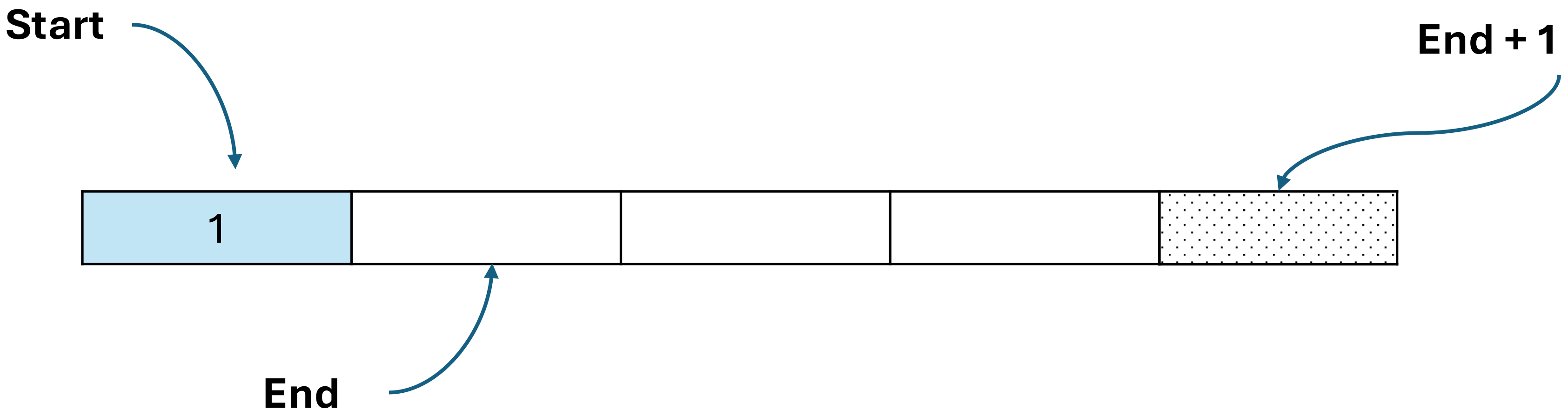
`lvalue or rvalue`

Pre C++11 : How Did `std::vector` work?

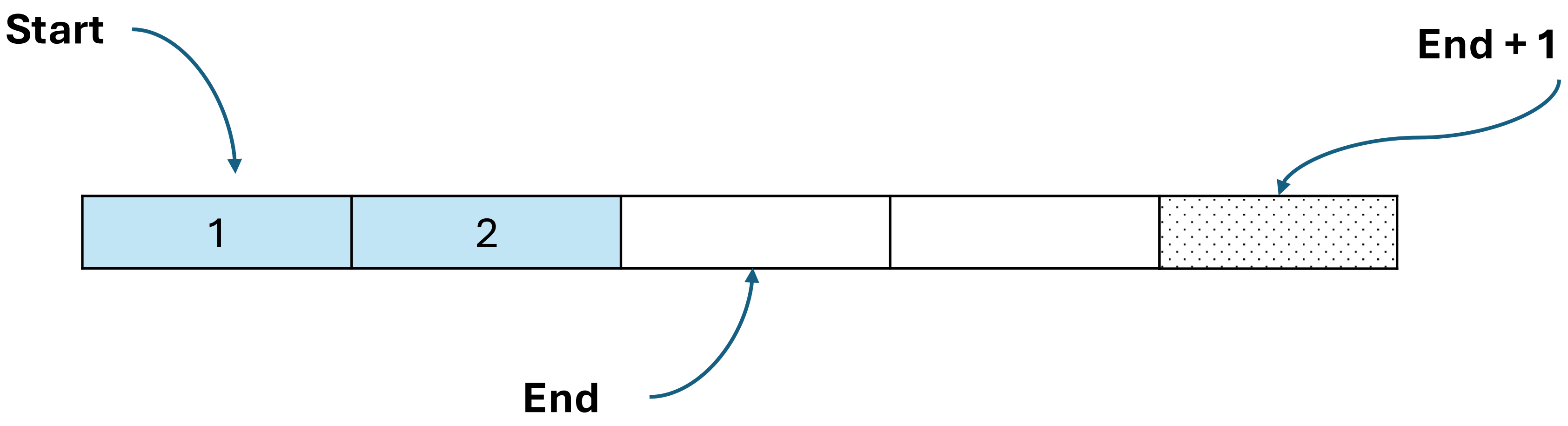
Pre C++11: How does a `std::vector` work?



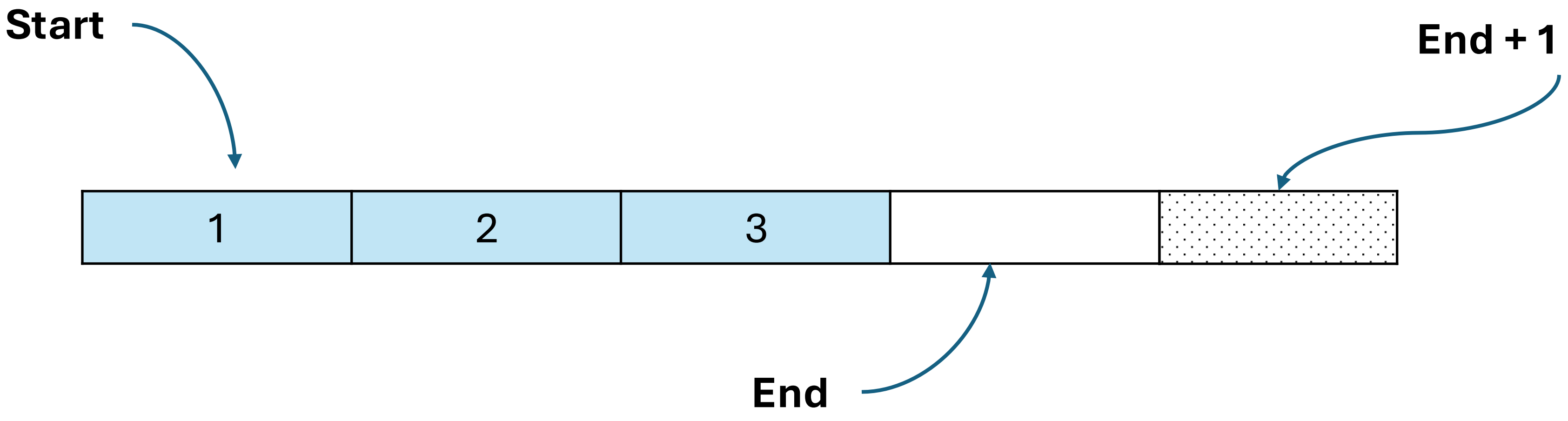
Pre C++11: How does a `std::vector` work?



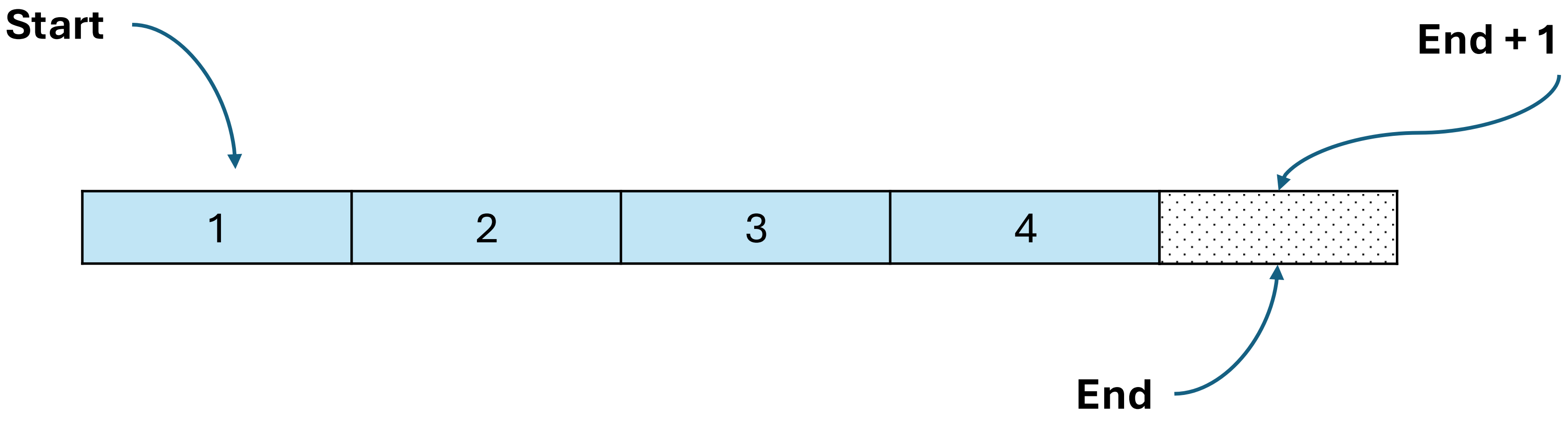
Pre C++11: How does a `std::vector` work?



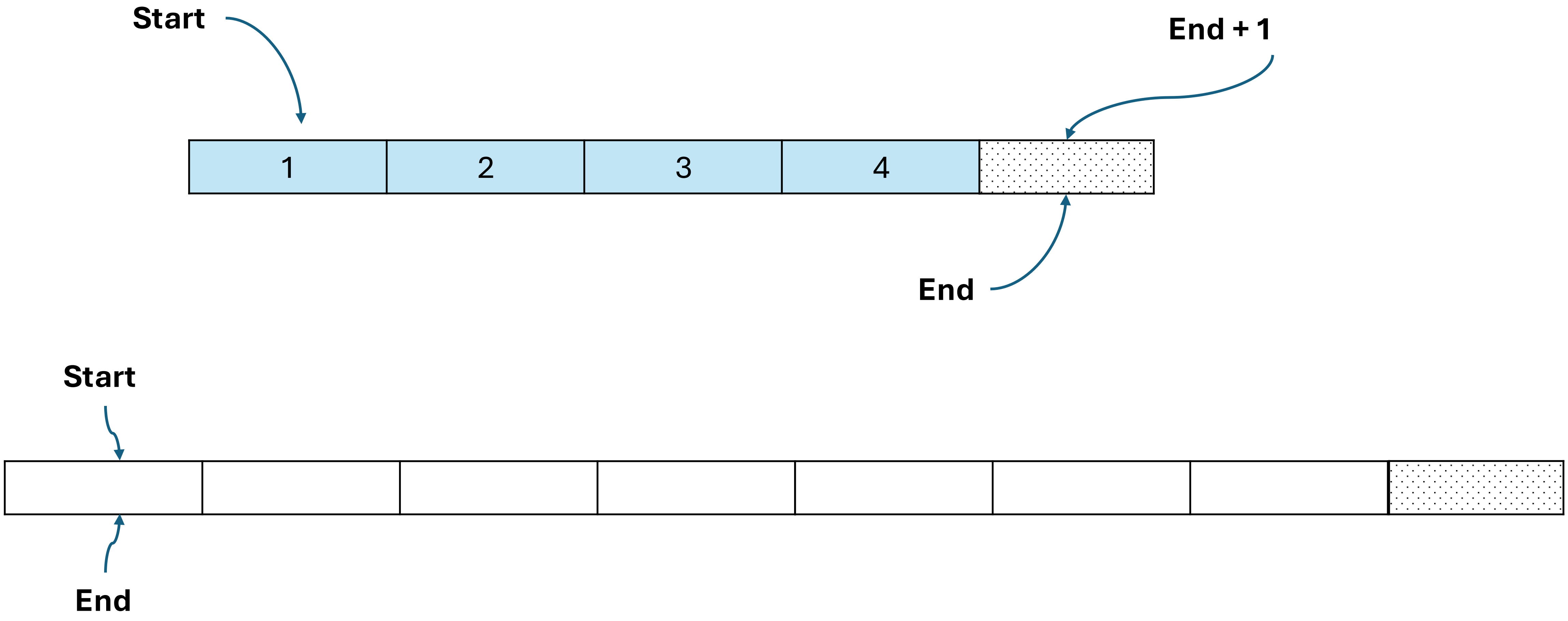
Pre C++11: How does a `std::vector` work?



Pre C++11: How does a `std::vector` work?

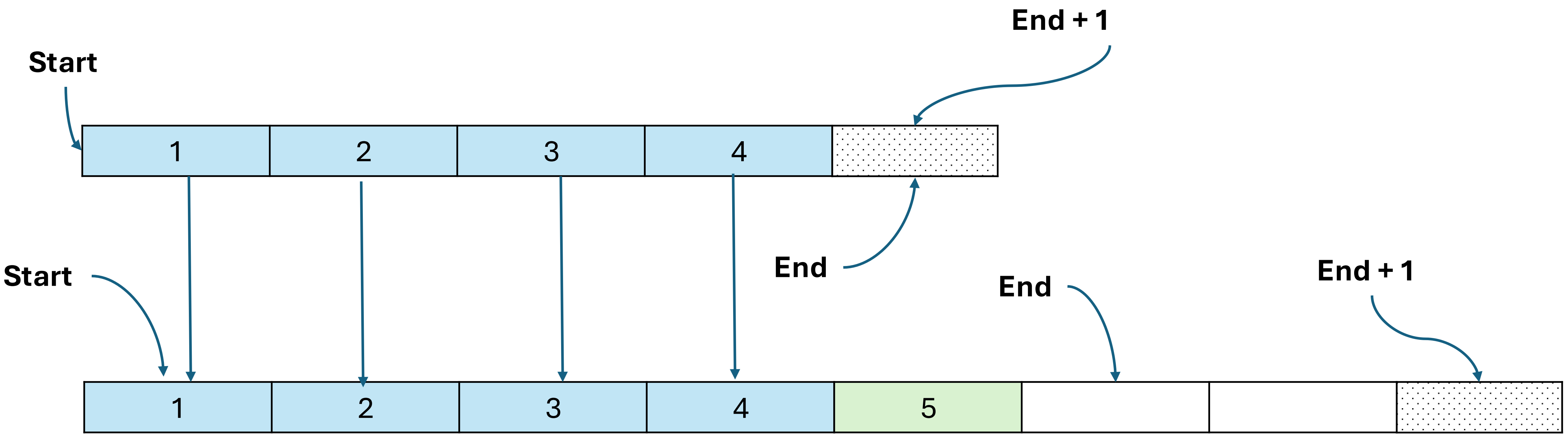


Pre C++11: How does a `std::vector` work?



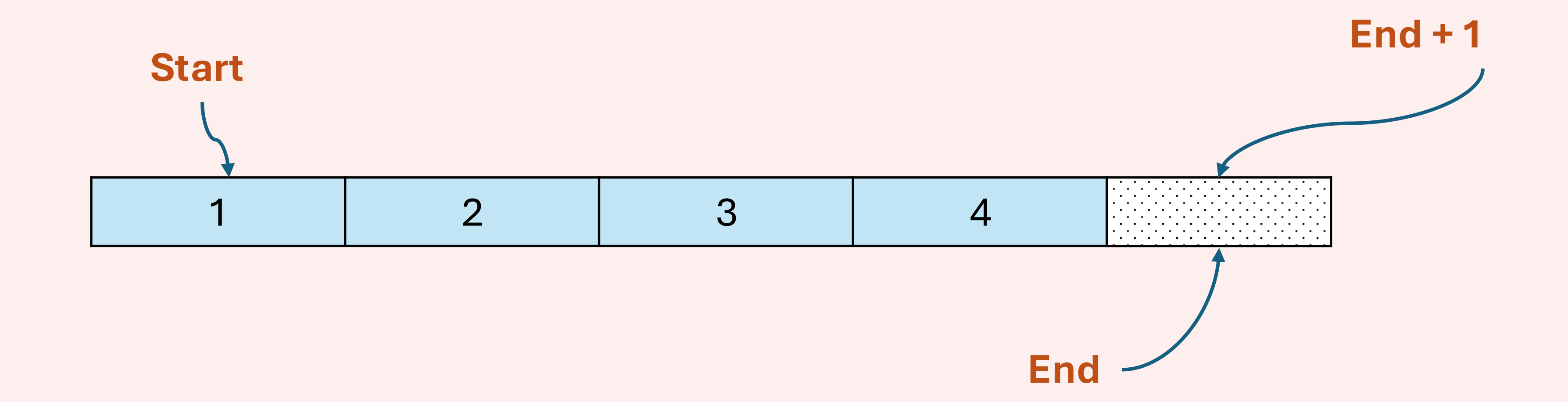
Reallocation

Pre C++11: How does a `std::vector` work?



expenseive copies

Pre C++11: How does a `std::vector` work?



- Copied Elements are now destroyed immediately after copy
- Theoretically, we could reuse the resources but there was no existing way in the language we could do this

Pre-C++11 : Value Categories & references

Value Categories

- Lvalues
  - Have identity and are long lived
  - `x , arr[i], …`

- Rvalues
  - Temporaries
  - `42, std::string(“hello”)`

Reference Type available

T&

- Lvalue reference
- Binds only to lvalues

```
int x = 42;
int &lvr = x;
```

- Can bind to rvalues only if **const T&**

```
int &lvr = 42;
// error: cannot bind rvalue to lvalue reference
```

Pre-C++11 Rvalue Binding Problems

```
void f(std::string); // copies rvalue
void f(const std::string&); // binds, but no mutation
void f(std::string&); // error: won't bind to rvalue
```

The two-category system was too limiting for modern C++ needs

The two problems pre C++11

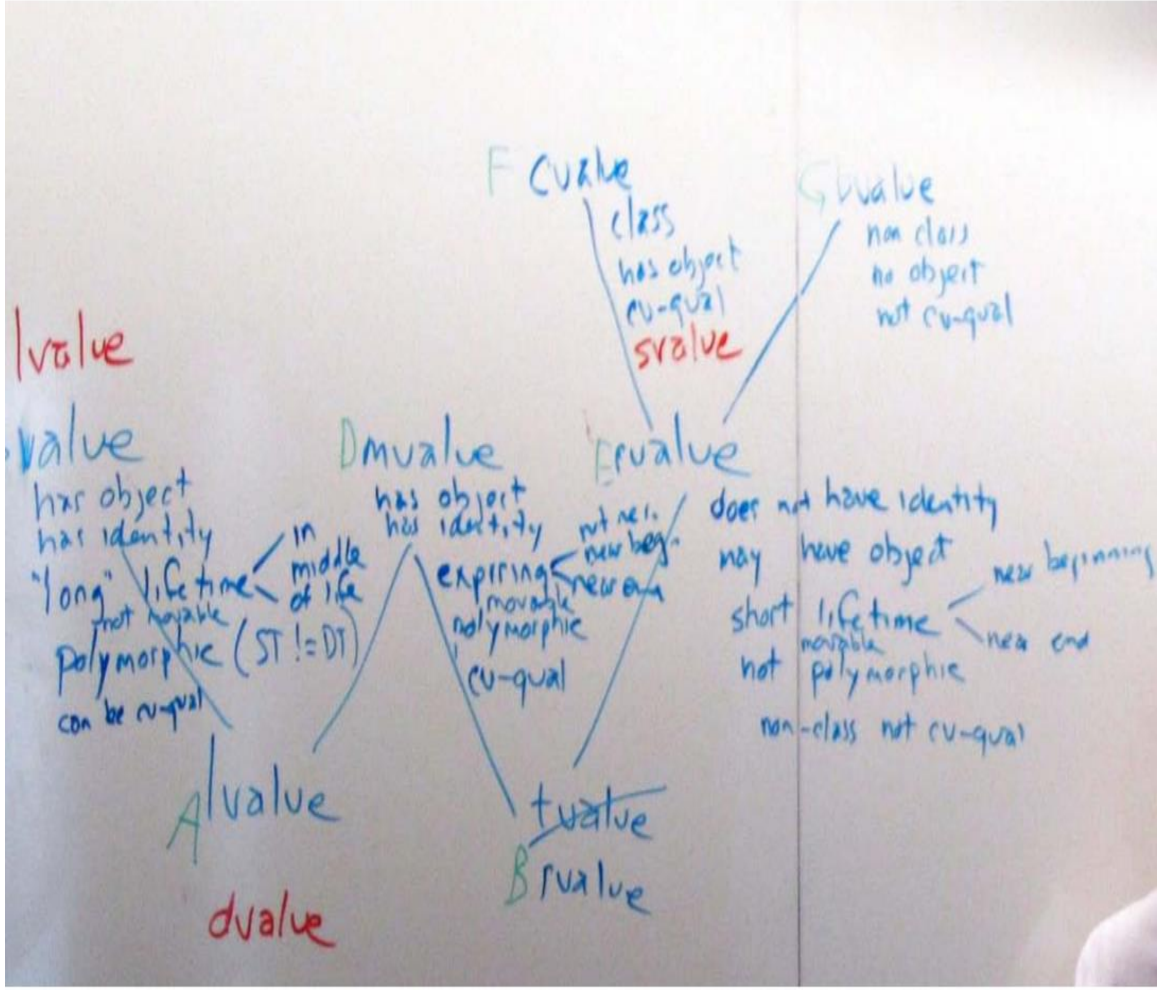
Cannot mark an object expired!

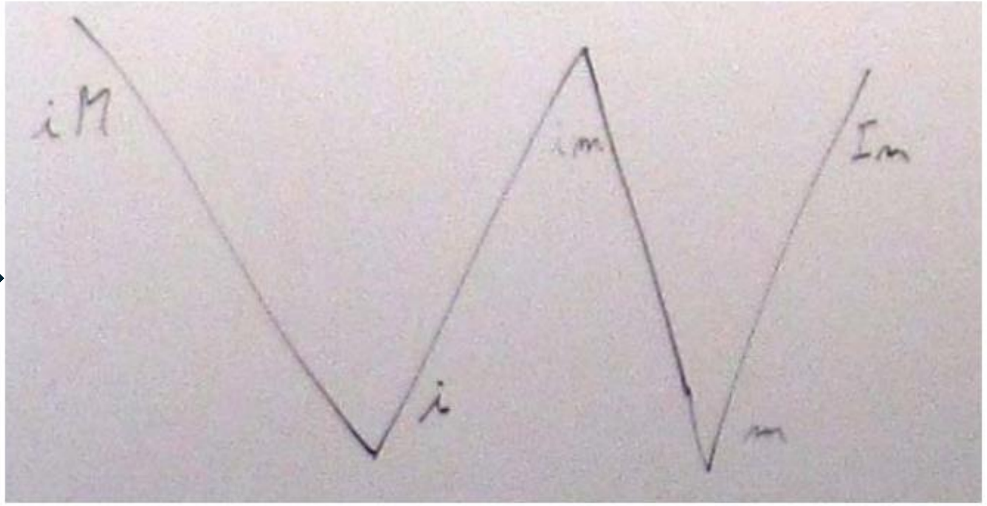
Type system had no way to bind and modify rvalues

Problem 1

Have a way to express that an object will soon expire

Trying to solve these problms





Value Categories Signify: Two Properties

Identity

Does the expression refer to an object with persistent identity?

"Can I take its address with &"

✓ x (has identity)
✗ 42 (no persistent identity)
identity)

Movable

Can we safely move resources from this expression?

"Is it safe to steal from?"

✓ func() (can move)
✗ z (can't move)

The Key Insight

These two properties are **independent!**

2 properties × 2 values

4 combinations possible

The 5 Value Categories

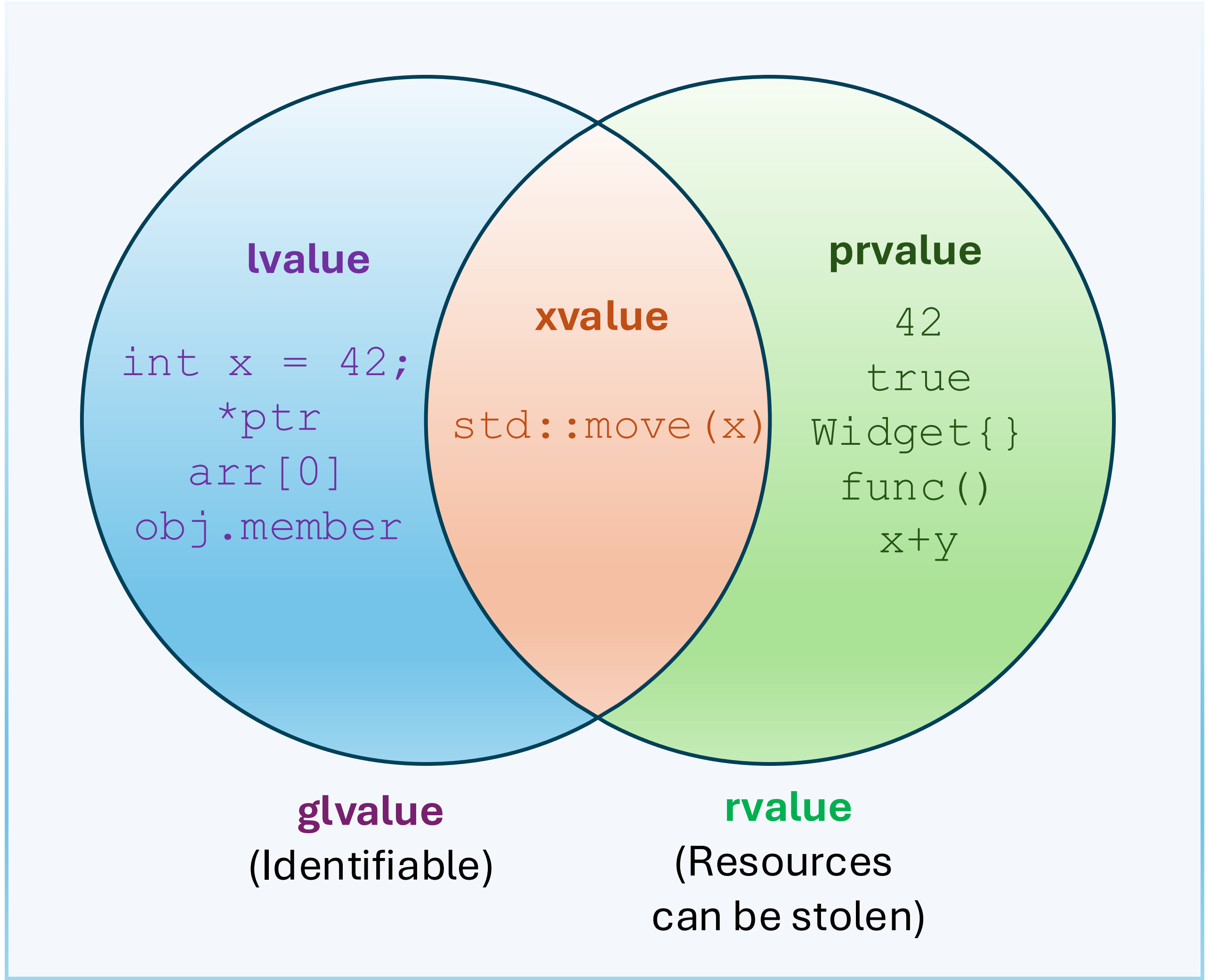
Properties	Identifiable	Not Identifiable
Not Movable	iM lvalue	Ignore (no such thing)
Movable	im xvalue	Im prvalue

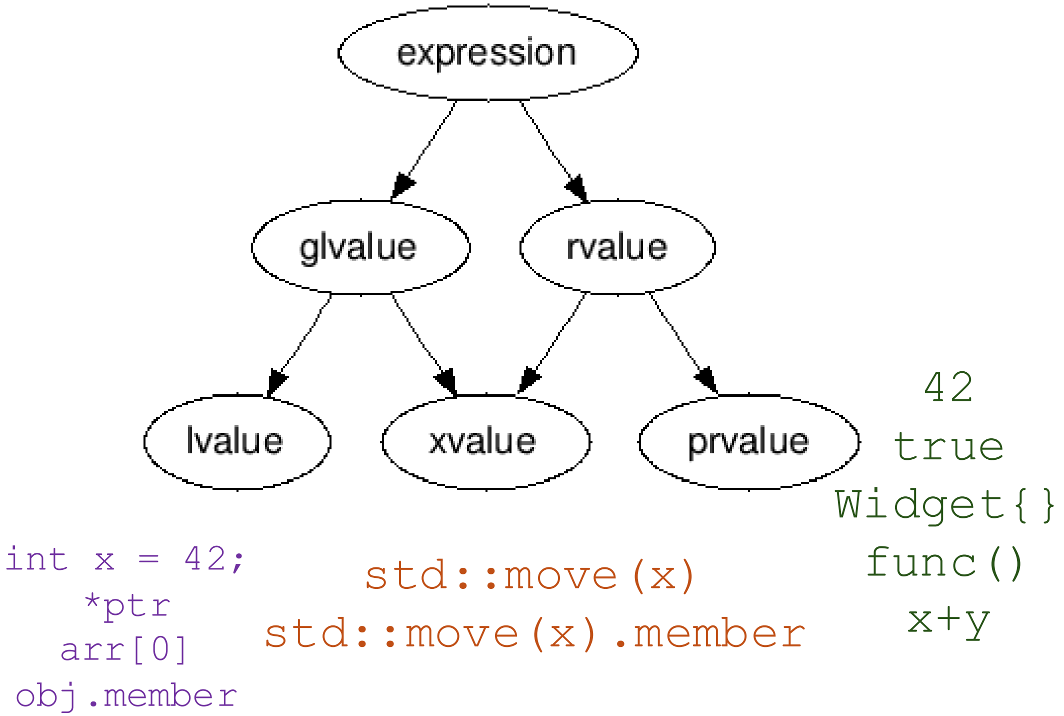
glvalue
(Generalized lvalue)

rvalue
(Right value)

💡The iM taxonomy: i = identifiable, m = movable (uppercase = false)

Other popular representations





lvalue Examples

```cpp
int x = 42;
a = b
++a
*p
a[n]
"hello"

void foo() {}
void baz()
{
    //'foo' is lvalue
    // You can take its address
  void (*p)() = &foo;
 }

 // Expression returning a
 int & get_ref(); //get_ref() is lvalue
```

Has Identity

- ✓ Refers to a memory location
- ✓ Can take its address with &
- ✓ Persists beyond the expression
- × cannot be moved from

Key Insight

If you can take the address of an expression, it's likely an lvalue.

prvalue Examples

```
//Any literal other than a string literal
42, true, nullptr

// any function call whose return type is non-reference
get_sum(a, b), str.substr(1, 2)

// Increments returns temporary(prvalue) with old value of a
a++

// Results in a prvalue - Has no identity or address
a + b

// Can't do &&a
&a

this

// prvalue - unnamed lambda closure type
[](int x){ return x * x; };
```

No Identity

- × Cannot take its address
- ✓ Can be moved from
- ✓ Temporary values
- ✓ Literals and computed results

Key Insight

prvalues are "pure" temporaries - they exist only for the duration of the expression.

xvalue Examples

```
std::string s = "hello";
std::move(s)

//std::string("hello") is prvalue but .substr(0, 4) is xvalue
std::string("hello").substr(0, 4);

//member access of xvalue is also xvalue
std::move(obj).member
```

⧗ About to Expire

- ✓ Has identity (like lvalue)
- ✓ Can be moved from (like rvalue)
- ✓ Result of std::move
- ✓ Enables move semantics

Key Insight

xvalues are the bridge between lvalues and move semantics.

Problem 2

How do we bind exclusively to these expiring values so as to steal the resources

Meet **T&& - Rvalue reference**

T&&

Syntax: `type&& var`

Binds to: rvalues (xvalues and prvalue)

Since: C++11

Purpose:

1. Provisions stealing of resources
2. Move semantics & perfect forwarding

```
// Bind prvalue (literal) to T&&
int &&r1 = 42;

// Bind function return (prvalue)
struct A { /*...*/ };
A CreateA();
A &&rr = CreateA();

// Bind subobject of temporary (xvalue)
std::string &&str =
        std::string("Hellooo").substr(0, 4);
```

const T&& - Binds to rvalues (read-only) – No Use in practice

Overload Resolution Priority

Value Category	T& (Alias)	const T& (Read only)	T&& (Steal me)	const T&& (Not used)
lvalue (Named object)	✓ 1st	✓ 2nd	✗ NO	✗ NO
xvalue (std::move(x))	✗ NO	✓ 2nd	✓ 1st	✓ 3rd
prvalue (Literal/temp)	✗ NO	✓ 2nd	✓ 1st	✓ 3rd

T&& :
- Bind to rvalues (Both xvalues and prvalues)
- Does not Bind to lvalues

Move Semantics

- Idea : Reuse internals of temporary/moved from object
- Main reason for having rvalue references
- `std::move` **marks an object as Expiring**

std::move – What does it really do?

```
template <typename T>
constexpr std::remove_reference_t<T>&& move(T&& t) noexcept {
    return static_cast<std::remove_reference_t<T>&&>(t);
}
```

- **Casts to an rvalue**, enabling move semantics
- **Does not move anything by itself** — it just *allows* moving

Move Constructor

```cpp
class Buffer {
    char* data;
    size_t size;
public:
    Buffer(size_t sz) : size(sz), data(new char[sz]) {}

    Buffer(const Buffer& other)
      : size(other.size), data(new char[other.size]) {
        std::copy(other.data, other.data + size, data);
    }

    Buffer(Buffer&& other) noexcept
        : size(other.size), data(other.data) {
        other.data = nullptr;
        other.size = 0;
    }

    ~Buffer() {
        delete[] data;
    }
};
```

`T&&` + `xvalue` = Problem Solved!

We solved both the problems

1. Express that values are going to expire – **xvalue**
2. Exclusively catch mutable rvalues – **T&&**

Quiz time again...

```
class Buffer {
  char *data;
  std::size_t size;
public:
  Buffer(std::size_t);
  Buffer(const Buffer&);
  Buffer(Buffer&&);
  ~Buffer();
}
```

```
const Buffer CreateBuffer() {
  return Buffer{42};
}
```

```
1  auto b1 = Buffer{42};                    // 1 new
2  auto b2 = std::move(b1);                 //0 new, 1 move
3  auto b3 = std::move(Buffer{42});         //1 new, 1 move
4  auto b4 = CreateBuffer();                // 1 new
5  auto b5 = std::move(CreateBuffer());     // 1 new, 1 copy
```

Value Category	T& (Alias)	const T& (Read only)	T&& (Steal me)	const T&& (Not used)
lvalue	✓ 1st	✓ 2nd	✗ NO	✗ NO
xvalue	✗ NO	✓ 2nd	✓ 1st	✓ 3rd
prvalue	✗ NO	✓ 2nd	✓ 1st	✓ 3rd

T&& when T is a template parameter

T&&: Not Always an Rvalue Reference

T&& in deduced context is referred to as **universal reference**

```
template <typename T>
void f(T&& x); // T is deduced, x is a universal reference

void g(int x); // int is fixed, not deduced
```

A **deduced type** is one where the compiler figures out T based on the argument

Reference Collapsing

What happens when two references Combine?

```
template <typename T>
void f(T&& x);

int x = 10;
f(x);
f(std::move(x));
```

Declared (T&, T&&)	**Deduced (int &, int &&)**	**Result**
&	&	&
&	&&	&
&&	&	&
&&	&&	&&

Declared = what you write (e.g. T&, T&&)
Deduced = what the compiler infers for T
Result = final type after reference collapsing

Reference Collapsing example

```
template <typename T>
void foo(T&& param); // param is a Universal Reference
```

Argument Type	T deduced as	T&& becomes
`foo(3)` 3 is prvalue	`int`	`int&&`
`foo(x)` (x is lvalue)	`int&`	`int&`
`foo(std::move(x))` std::move(x) is xvalue	`int&&`	`int&&`

Universal references can bind both to lvalues and rvalues

Use `std::forward` for universal references

Why `std::forward`?

- A universal reference (T&&) can bind to both lvalues and rvalues.
- std::forward preserves value category

```
template <typename T>
void Wrapper(T&& arg) {
    callee(arg); // Bad - always lvalue — copy happens
    callee(std::forward<T>(arg)); // Good - preserves value category
}
```

Efficient code with `std::forward`

```
class Pool {
  std::vector<Resource> resources;
public:
  void addResource(const Resource& r) {
    resources.push_back(r);
  }
  void addResource(Resource&& r) {
    resources.push_back(std::move(r));
  }
};
```

```
template <typename T>
void addResource(T&& r) {
  resources.push_back(std::forward<T>(r))
}
```

Best Practices surrounding value categories

Don't `std::move` return values - Trust RVO

```cpp
A Create() {
	A a;
	// RVO kicks in, no move/copy
	return a;
}
```

```cpp
A Create() {
	A a;
	// Disables copy elision
	return std::move(a);
}
```

When a function returns a **prvalue**, the object is constructed **directly in the caller's storage**.

Pass by value and then move

```cpp
void setName(std::string s)
{
   // moves if possible
   name_ = std::move(s);
}
```

```cpp
void setName(const std::string& s)
{
   name = s; // always copies
}
```

```cpp
setName(std::string("hello"));
```

Rvalues are efficiently handled when passed by value and moved

Only use std::forward<T>(x)
ONLY
if T is a deduced type

```
template <typename T>
void wrapper(T&& x)
{
  // perfect forwarding
  process(std::forward<T>(x));
}
```

```
void misuse(std::string& s)
{
  // forces move from lvalue
  process(std::forward<std::string>(s));
}
```

Never use std::forward<T>(x) unless T is a deduced template parameter

Mark move operators `noexcept`

```cpp
struct A
{
  A(A&& other) noexcept { /* ... */ }
};
```

```cpp
struct A
{
  A(A&& other) { /* ... */ }
};
```

Always mark move constructors and assignments noexcept so STL containers can safely move your objects.

Prefer `std::move_if_noexcept` for strong exception safety

```
vec.push_back(std::move_if_noexcept(obj));
```

```
vec.push_back(std::move(obj));
```

- STL containers rely on this to **maintain strong exception safety.**

Key Takeaways

Core Concepts

Value Categories
Every expression has a type and value category (lvalue, xvalue, prvalue)

Move Semantics
Transfer resources instead of copying for better performance

Perfect Forwarding
Preserve value categories when passing arguments through templates

Practical Guidelines

When to Move
Use std::move when you're done with an object, not on returns

When to Forward
Use std::forward in templates with universal references

Trust the Compiler
RVO beats move, move beats copy - let the compiler optimize

Understand
Value categories

Optimize
With move semantics

Forward
Preserve categories

Avoid
Common pitfalls

Value categories are the foundation of modern, efficient C++

Thank you!

Questions Please!