# Understanding C++ value categories

Belgian C++ User Group - BeCPP, June 24, 2021

Kris van Rens

# What’s ahead?

- What are value categories?

*(questions)*

- Value categories in the wild

# A little bit about me

🤓 👩‍👩‍👧‍👦 🎸 🏃 🧗

kris@vanrens.org

# The premise and goals

# Quiz

```cpp
struct Number {
  int value_ = {};
};

class T {
public:
  T(const Number &n) : n_{n} {}

  T(const T &) { puts("Copy c'tor"); }

  Number get() { return n_; }

private:
  Number n_;
};
```

```cpp
static T create(Number &&n) {
  return T{std::move(n)};
}

int main() {
  T x = T{create(Number{42})};

  return x.get().value_;
}
```

What's the output?

# What are value categories?

# It all starts with...

# ...expressions!

*Value categories are **not about objects or class types**, they are about **expressions**!*

# I mean, seriously...

# ...expressions!

# What is an expression?

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

# Expression outcome

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

# Example expressions (1)

```
42  // Expression evaluating to value 42

17 + 42  // Expression evaluating to value 59
```

```
int a;

a = 23  // Expression evaluating to value 23
a + 17  // Expression evaluation to value 40
```

```
static_cast<float>(a)  // Expression evaluating to floating-point value 23.0f
```

# Example expressions (2)

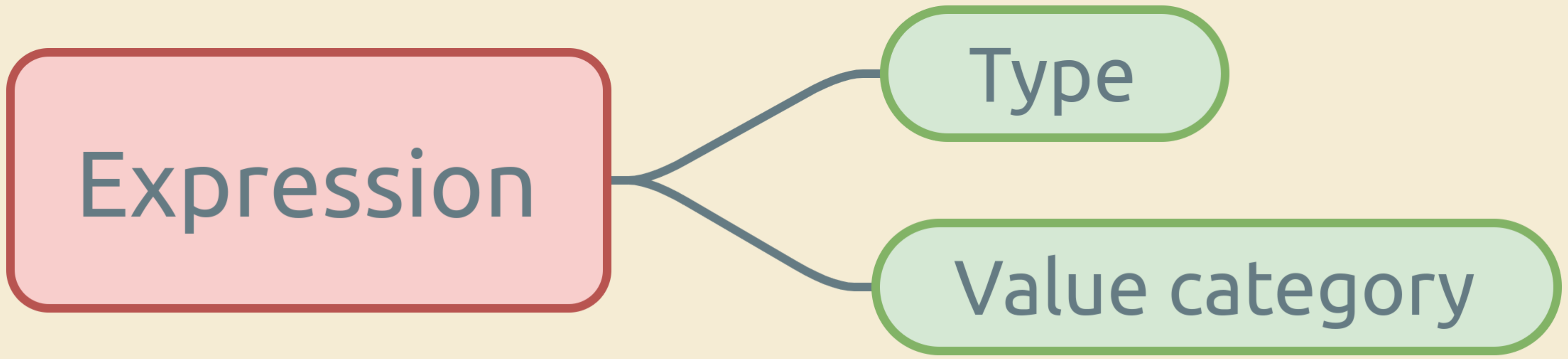
```
int a;

sizeof a  // Expression evaluating to the byte size of 'a'
          // Id-expression 'a' is unevaluated
```

```
[]{ return 3; }  // Expression evaluating to a closure
```

```
printf("Hi!\n")  // Expression evaluating to the number of characters written
                 // Result is often discarded, i.e. a 'discarded-value expression'
```

# Expressions in C++

In C++, each expression is identified by two properties:



# Primary value categories

`lvalue` – Locator value

`prvalue` – Pure rvalue

`xvalue` – eXpiring value

# But wait...there's more!

**`glvalue`** – General `lvalue`

**`rvalue`** – errrRrr..value ☠️

# Back to expressions

Value categories are organized based on expression properties:

1. Does it evaluate to an identity?
2. Can its result resources be safely stolen?

# Does it evaluate to an identity?

```cpp
int a;

a  // Has identity
```

```cpp
42                 // Has no identity
nullptr            // Has no identity
false              // Has no identity
[]{ return 42; }  // Has no identity
"Hi"               // Has identity
```

```cpp
std::cout  // Has identity
```

```cpp
a + 2      // Has no identity
a || true  // Has no identity
```

```cpp
a++  // Has no identity
++a  // Has identity
```

```cpp
static_cast<int>(a)  // Has no identity
std::move(a)         // Has identity
```

# Can its resources be safely stolen?

> *Expression result resources can be stolen if it evaluates to an anonymous temporary, or if the associated object is near the end of its lifetime.*

This was the main motivation for move semantics 🤔

# Can its resources be safely stolen?

```
std::string func()
{
  return "Steal me!";
}

std::vector<std::string> vec;

vec.push_back(func());
```

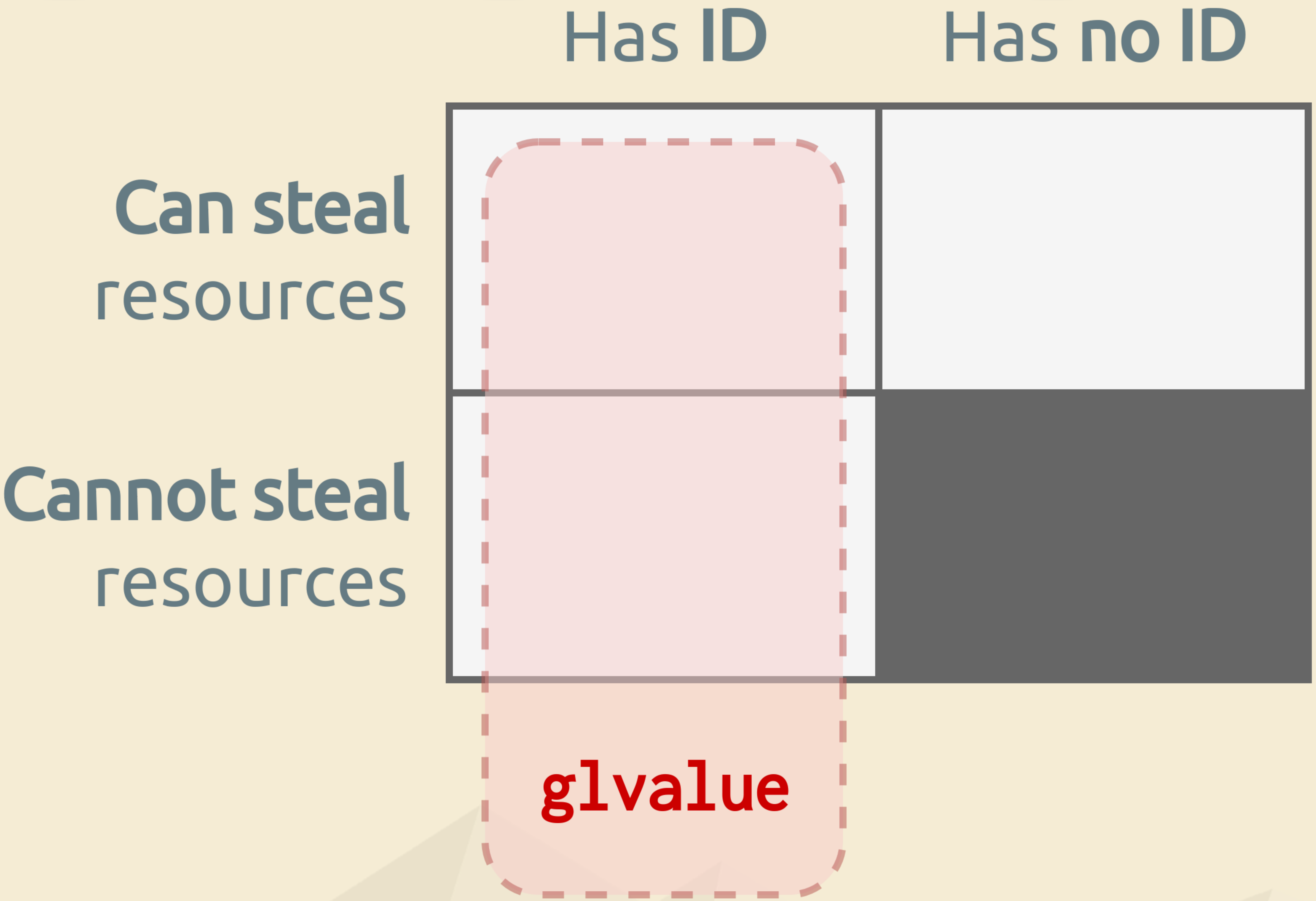
```
std::string x{"Steal me!"};

std::vector<std::string> vec;

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

# Let’s get organized!

| Has **ID** | Has **no ID** |
| --- | --- |
| | |
| | |

| | Has **ID** | Has **no ID** |
|---|---|---|
| **Can steal** resources | | |
| **Cannot steal** resources | | |

| | Has **ID** | Has **no ID** |
|---|---|---|
| **Can steal** resources | | |
| **Cannot steal** resources | | |


Has ID
Has no ID
Can steal resources
Cannot steal resources
glvalue

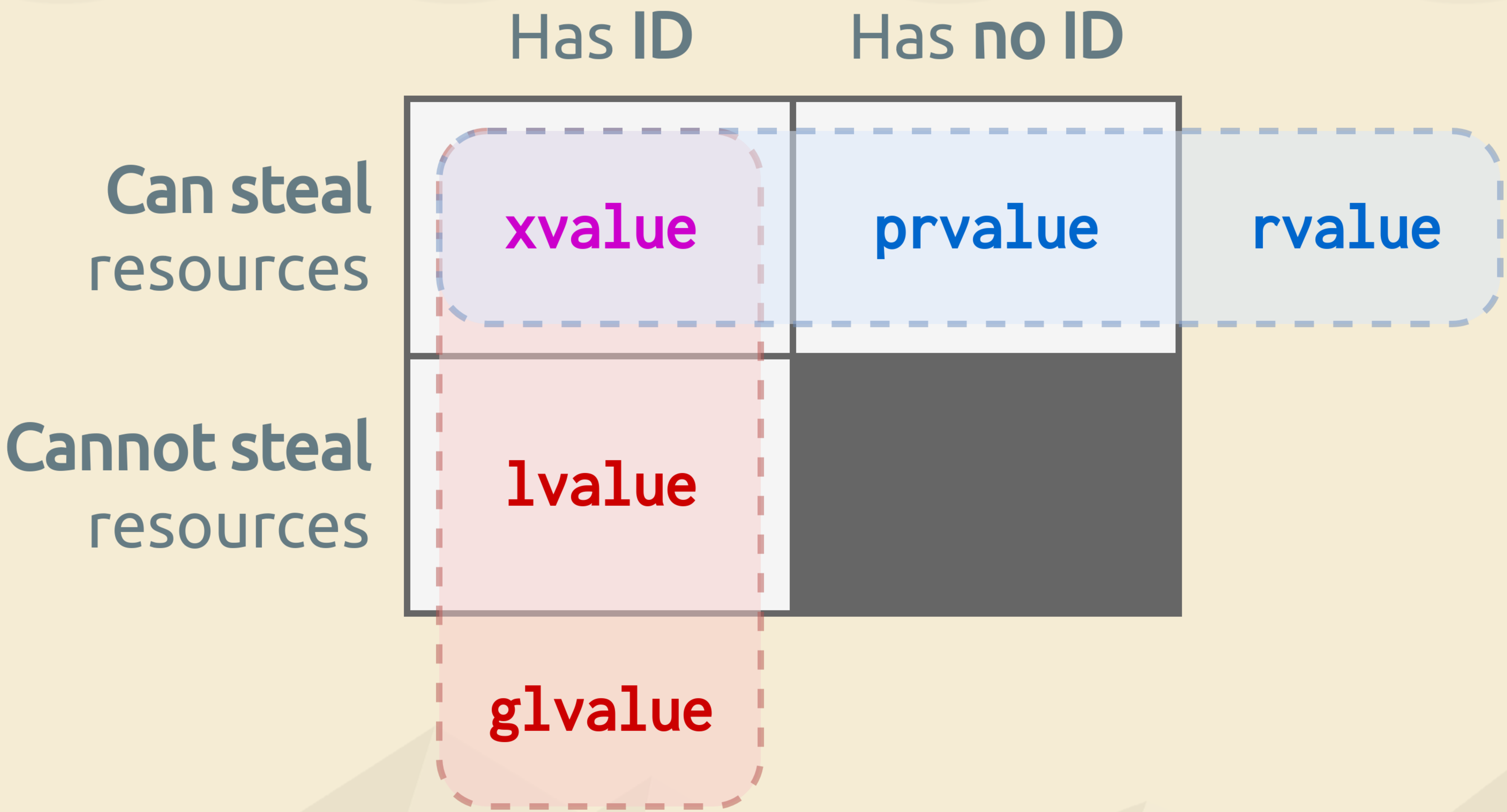
| | Has ID | Has no ID |
|---|---|---|
| **Can steal** resources | | prvalue |
| **Cannot steal** resources | | |

glvalue

| | Has ID | Has **no** ID |
|---|---|---|
| **Can steal** resources | `xvalue` | `prvalue` |
| **Cannot steal** resources | | |

`glvalue`

| | Has **ID** | Has **no ID** |
|---|---|---|
| **Can steal** resources | `xvalue` | `prvalue` |
| **Cannot steal** resources | `lvalue` | |
| | `glvalue` | |

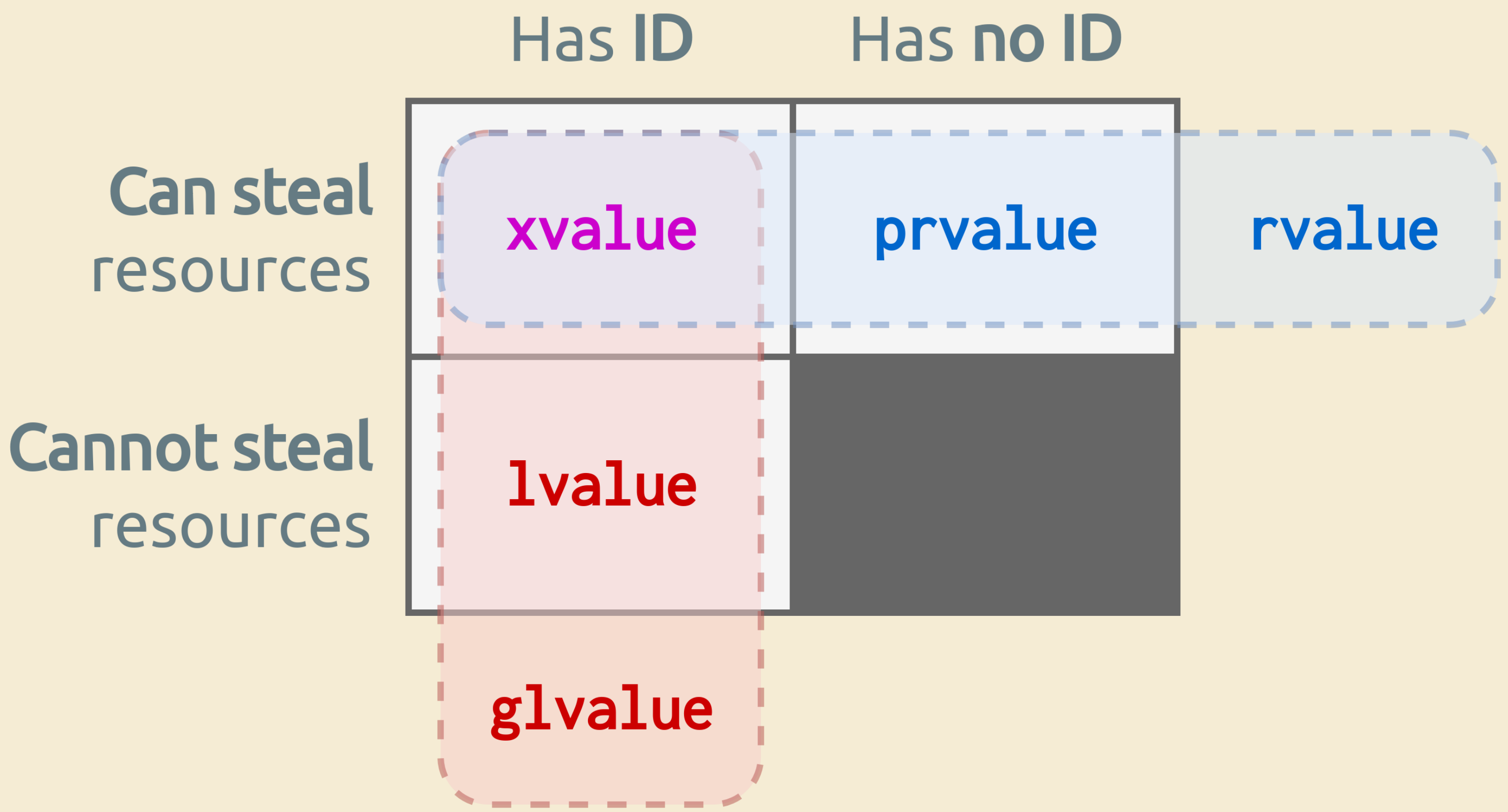

Has ID
Has no ID
Can steal resources
Cannot steal resources
xvalue
prvalue
rvalue
lvalue
glvalue

# Examples (1)

```
42   // prvalue
```

```
nullptr   // prvalue
```

```
"Hi there!"   // lvalue
```

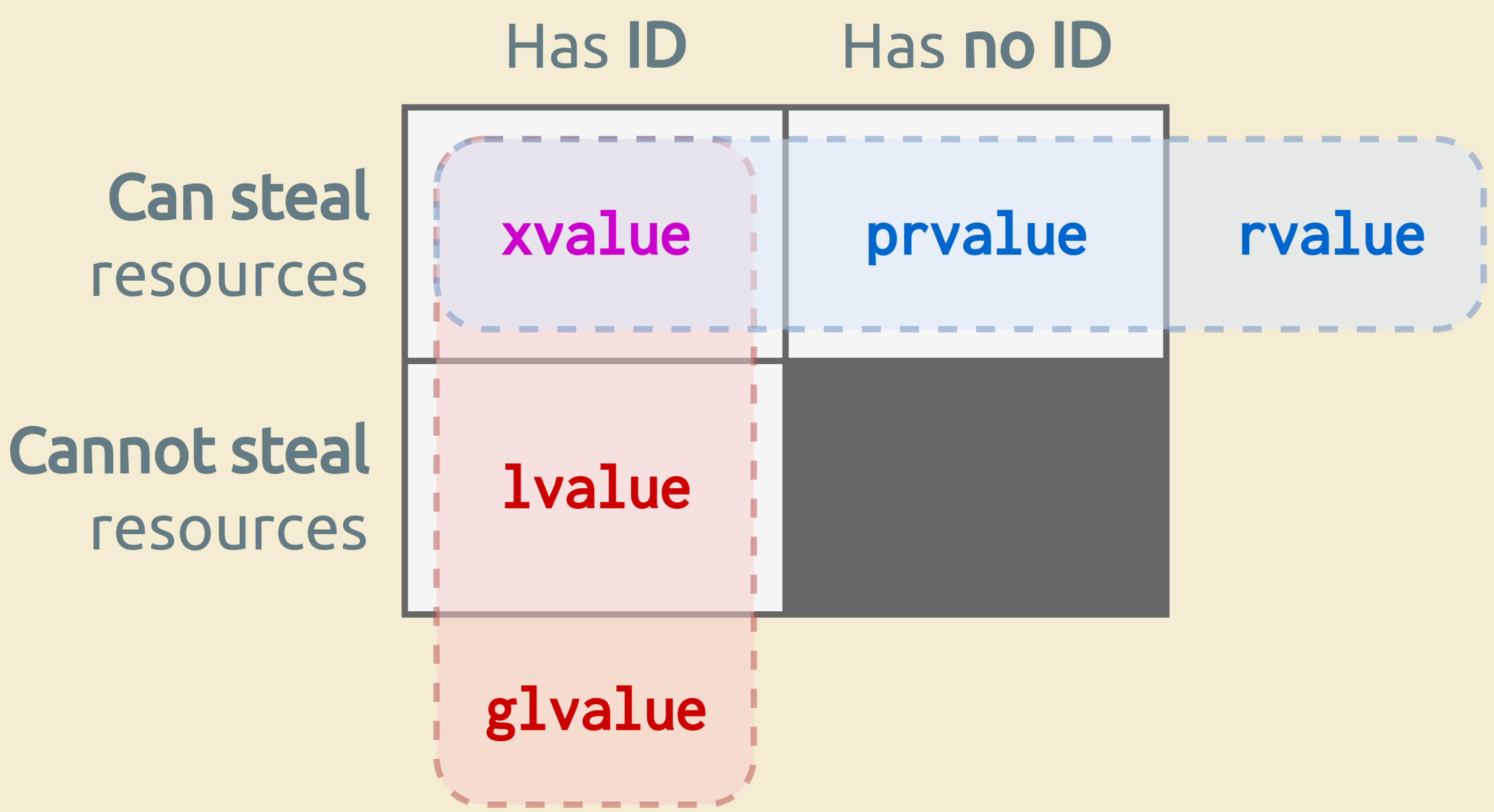


# Examples (2)

```
int x = 42;
```

```
++x   // lvalue
```

```
x++   // prvalue
```

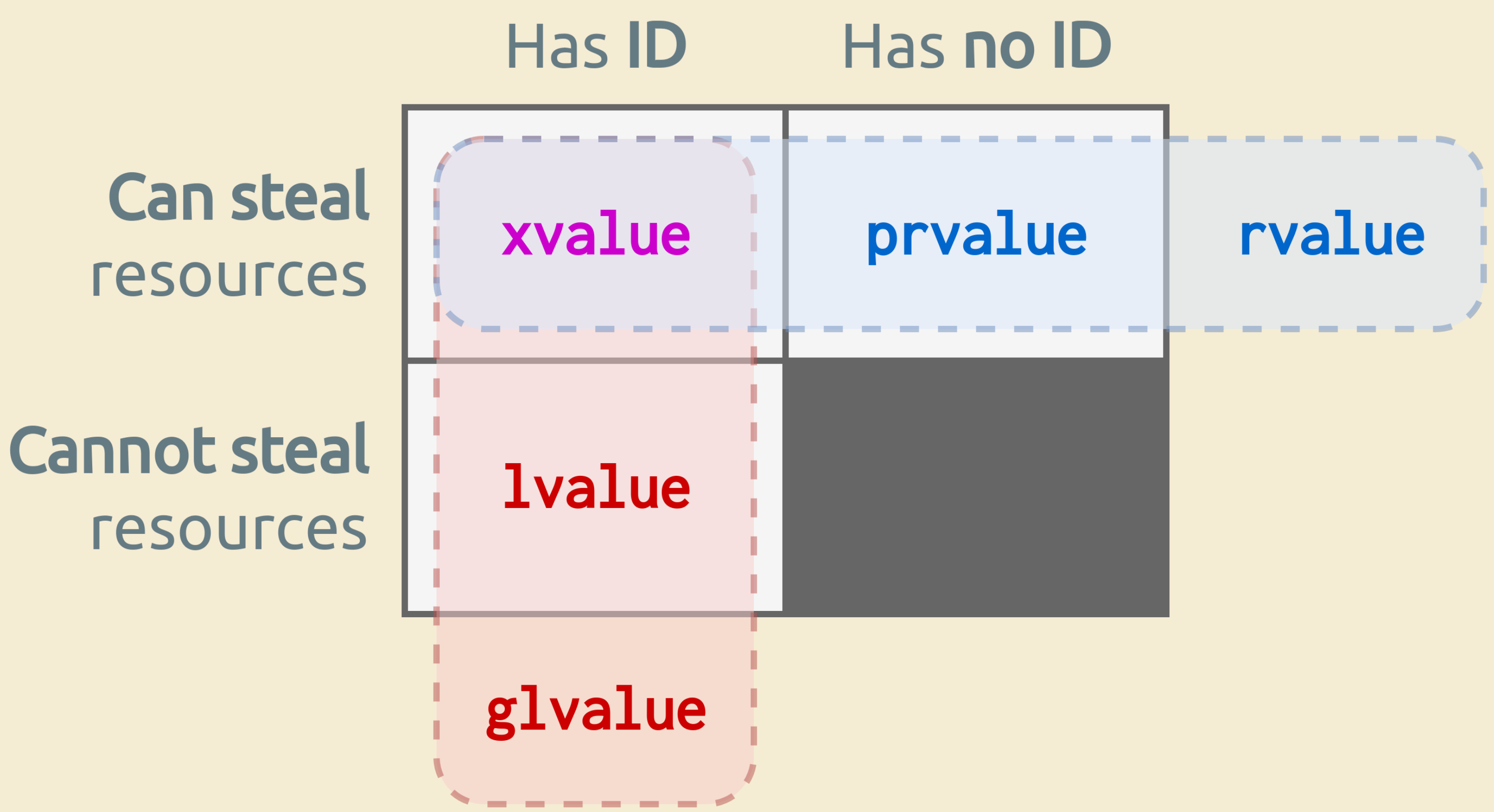


# Examples (3)

```
int x = 42;
```

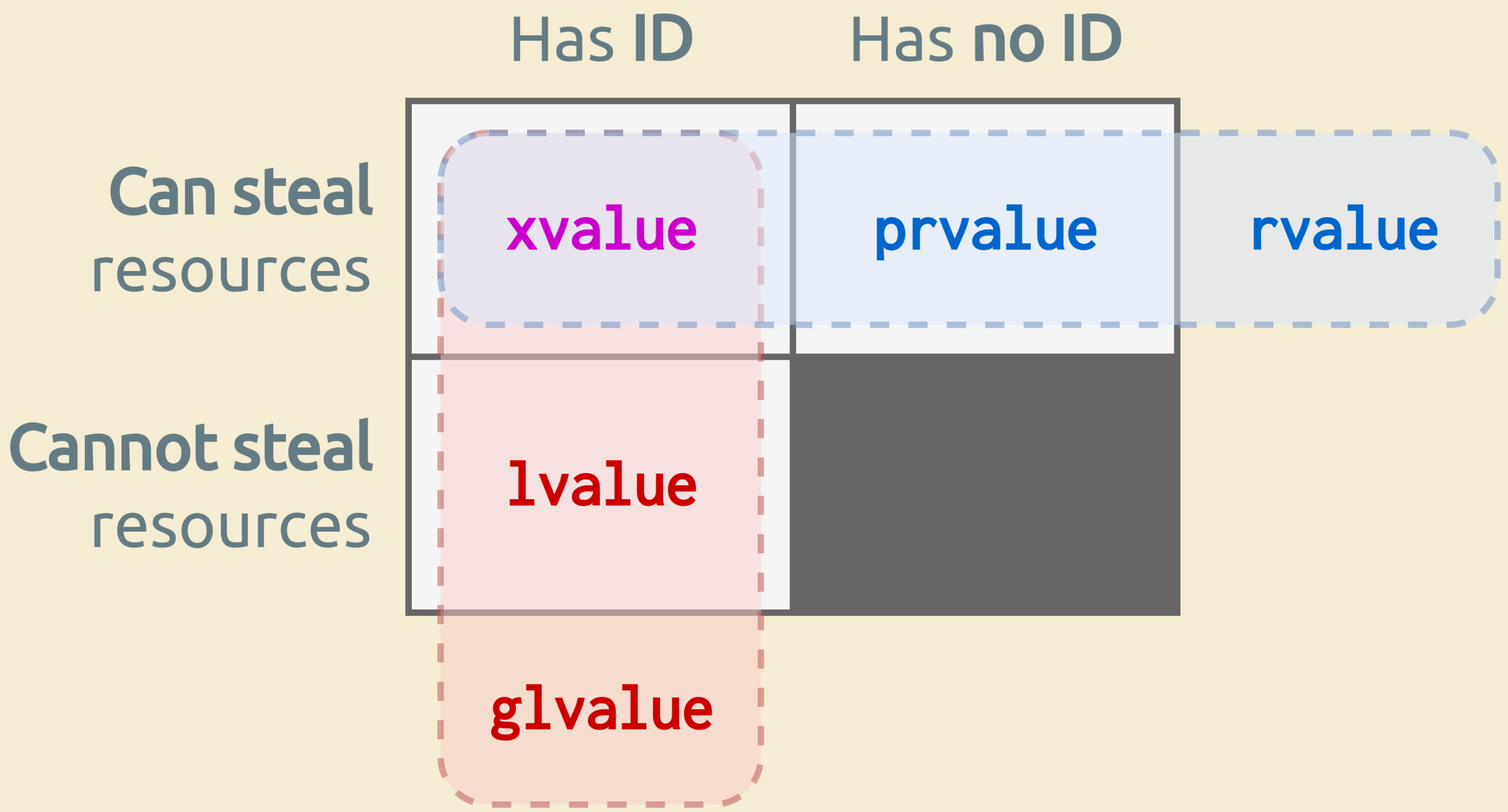
```
x   // lvalue
```

```
std::move(x)  // xvalue
```



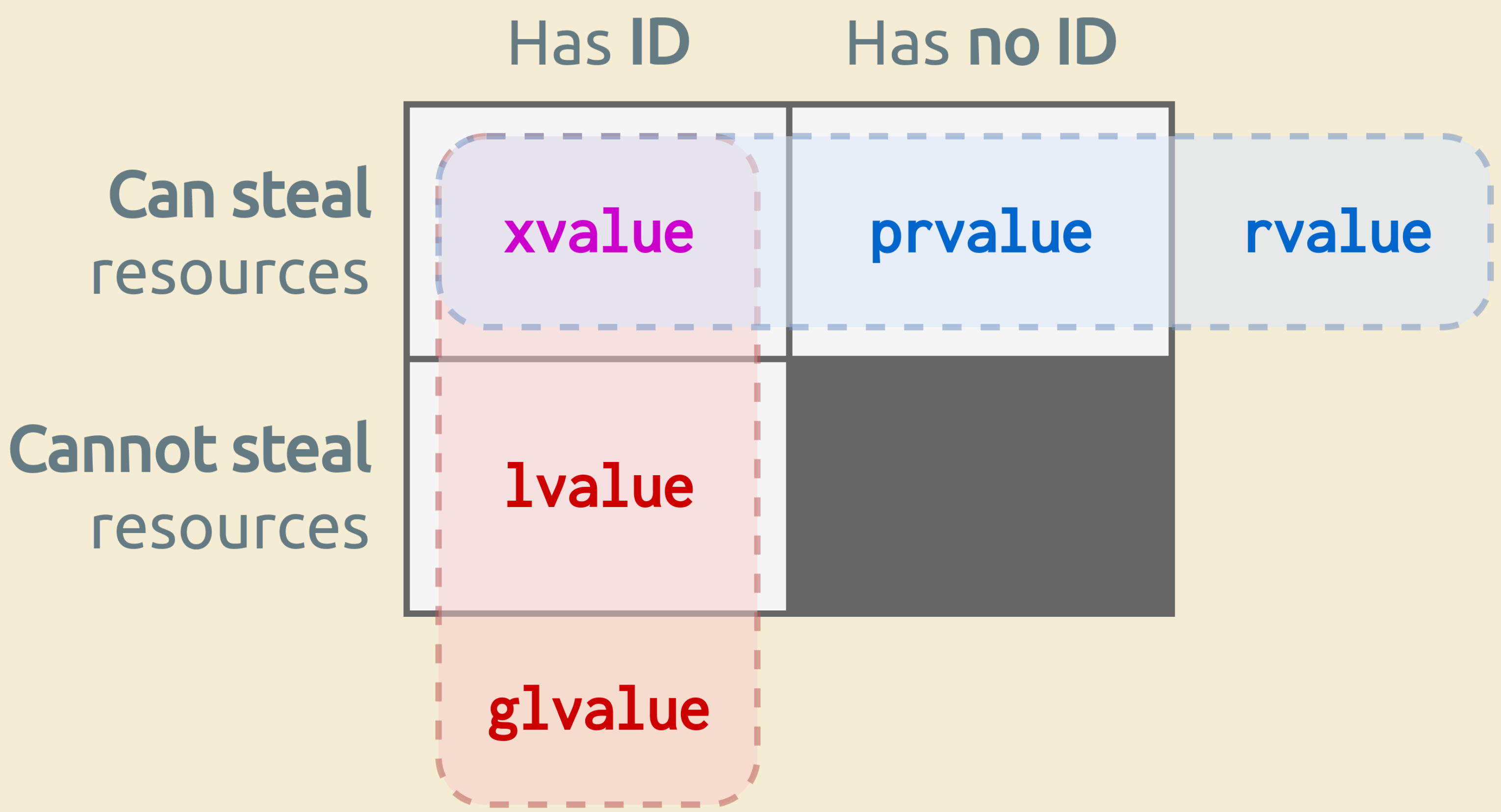
# Examples (4)

```cpp
void func(int &&arg)
{
  // 'arg' is an lvalue

  // 'std::move(arg)' is an xvalue
  other_func(std::move(arg));
}

func(42);  // '42' is a prvalue
```



# Examples (5)

```cpp
void func(int &arg);    // #1
void func(int &&arg);   // #2

int &&x = 42;

func(x);   // Which overload is called?
```




Expression `x` is an **`lvalue`**; so overload #1 is called

# A little side step: history

- CPL (1963) first introduced `lvalue` and `rvalue` concepts,
- Via BCPL and B came along C, keeping the definitions,
- C++ first followed the C definition up until C++03,
- C++11 introduced move semantics, changing it again.

Please forget the right-/left-hand notion for today’s definition.

# OK then. Now what?

- Communication: learn and be literate!
- Reading compiler errors effectively,
- In-depth knowledge on C++ is generally beneficial,
- Useful for understanding move semantics,
- Understanding copy elision and implicit conversions.

# Quiz revisited

```cpp
struct Number {
  int value_ = {};
};

class T {
public:
  T(const Number &n) : n_{n} {}

  T(const T &) { puts("Copy c'tor"); }

  Number get() { return n_; }

private:
  Number n_;
};
```

```cpp
static T create(Number &&n) {
  return T{std::move(n)};
}

int main() {
  T x = T{create(Number{42})};

  return x.get().value_;
}
```

What’s the output?

# Questions?

# Value categories in the wild

# Copy elision

*A section in the C++ standard that describes the elision (i.e. **omission**) of copy/move operations, resulting in zero-copy pass-by-value semantics.*

Restrictions apply 😞

# Copy elision

> ***Permits*** *elisions, it does not* ***guarantee****!*

Actual results depend on compiler and compiler settings.

# Copy elision in action

## C++ code:

```
T func()
{
  return T{};  // Create temporary
}

T x = func();  // Create temporary
```

## Possible output (1):

```
T()
T(const &)
~T()
T(const &)
~T()
~T()
```

No copy elision.

# Copy elision in action

## C++ code:

```
T func()
{
  return T{};  // Create temporary?
}

T x = func();  // Create temporary?
```

## Possible output (2):

```
T()
T(const &)
~T()
~T()
```

Partial copy elision.

# Copy elision in action

## C++ code:

```cpp
T func()
{
  return T{};
}

T x = func();
```

## Possible output (3):

```
T()
~T()
```

Full copy elision.

# Where can elisions occur?

- In the initialization of an object,
- In a `return` statement,
- In a `throw` expression,
- In a `catch` clause.

# Great stuff!

Truth is; compilers have been doing it for years.. 😏

# Copy elision since C++17

C++17 added **mandates** to the standard, informally known as:

- “Guaranteed copy elision”,
- “Guaranteed return value optimization”,
- “Copy evasion”.

A set of special rules for `prvalue` expressions.

# Guaranteed copy elision (1)

> *If, in an initialization of an object, when the initializer expression is a* **`prvalue`** *of the same class type as the variable type.*

```
T x{T{}}; // Only one (default) construction of T allowed here
```

# Guaranteed copy elision (2)

*If, in a return statement the operand is a **prvalue** of the same class type as the function return type.*

```
T func()
{
  return T{};
}

T x{func()}; // Only one (default) construction of T allowed here
```

# Under the hood

> *Under the rules of C++17, a `prvalue` will be used only as an **unmaterialized recipe** of an object, until actual materialization is required.*

> *A `prvalue` is an expression whose **evaluation initializes/materializes** an object.*

This is called a *temporary materialization conversion.*

# Temporary materialization

```
struct Person {
  std::string name_;
  unsigned int age_ = {};
};

Person createPerson() {
  std::string name;
  unsigned int age = 0;

  // Get data from somewhere in runtime..

  return Person{name, age};    // 1. Initial prvalue expression
}

int main() {
  return createPerson().age_;  // 2. Temporary materialization: xvalue
}
```

# Temporary materialization

*An implicit **prvalue** to **xvalue** conversion.*

Remember: **prvalue**s are not moved from!

# Temporary materialization

Occurs when:

- Accessing a member of a **prvalue**,
- Binding a reference to a **prvalue**,
- Applying `sizeof` or `typeid` to a **prvalue**,
- Etc.

# Temporary materialization in action

```
struct T { int value; };

T{}          // prvalue
T{}.value    // xvalue
```

```
auto x = std::string("Guaca") + std::string("mole").c_str();
//     ^                       ^                     ^
//     3                       2                     1
```

# C++17 copy/move elision

## = Copy elision + temporary materialization

😎

# Return Value Optimization

## AKA ‘RVO’

A variant of copy elision.

# Return Value Optimization

Two forms:

1. Unnamed RVO (URVO or simply RVO),
2. Named RVO (NRVO).

# Return Value Optimization

RVO

URVO

NRVO

These terms live outside the standard.

# Unnamed RVO (URVO)

Refers to the returning of temporary objects from a function.

***Guaranteed** by C++17 rules.*

# Named RVO (NRVO)

Refers to the returning of named objects from a function.

# NRVO in action

## The most simple example

```
T func()
{
  T result;

  return result;
}

T x = func();
```

```
T()
~T()
```

# NRVO in action

## Slightly more involved

```
T func()
{
  T result;

  if (something)
    return result;

  // ...

  return result;
}

T x = func();
```

```
T()
~T()
```

It still works

# NRVO is finicky though

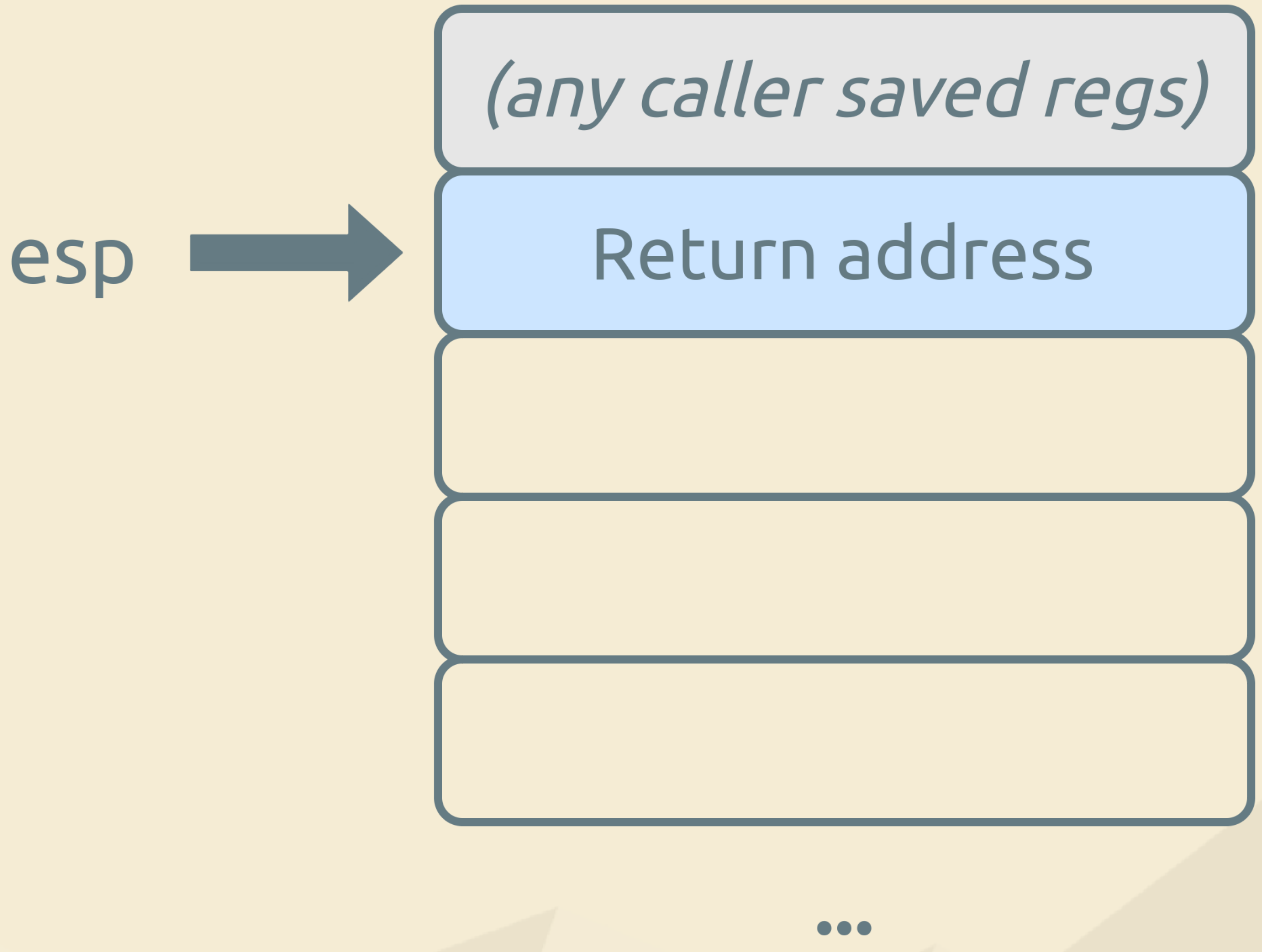
# Quick re-hash

## Function call mechanics

# No parameters, no return value

```
void func()
{
  // ...
}

func();
```

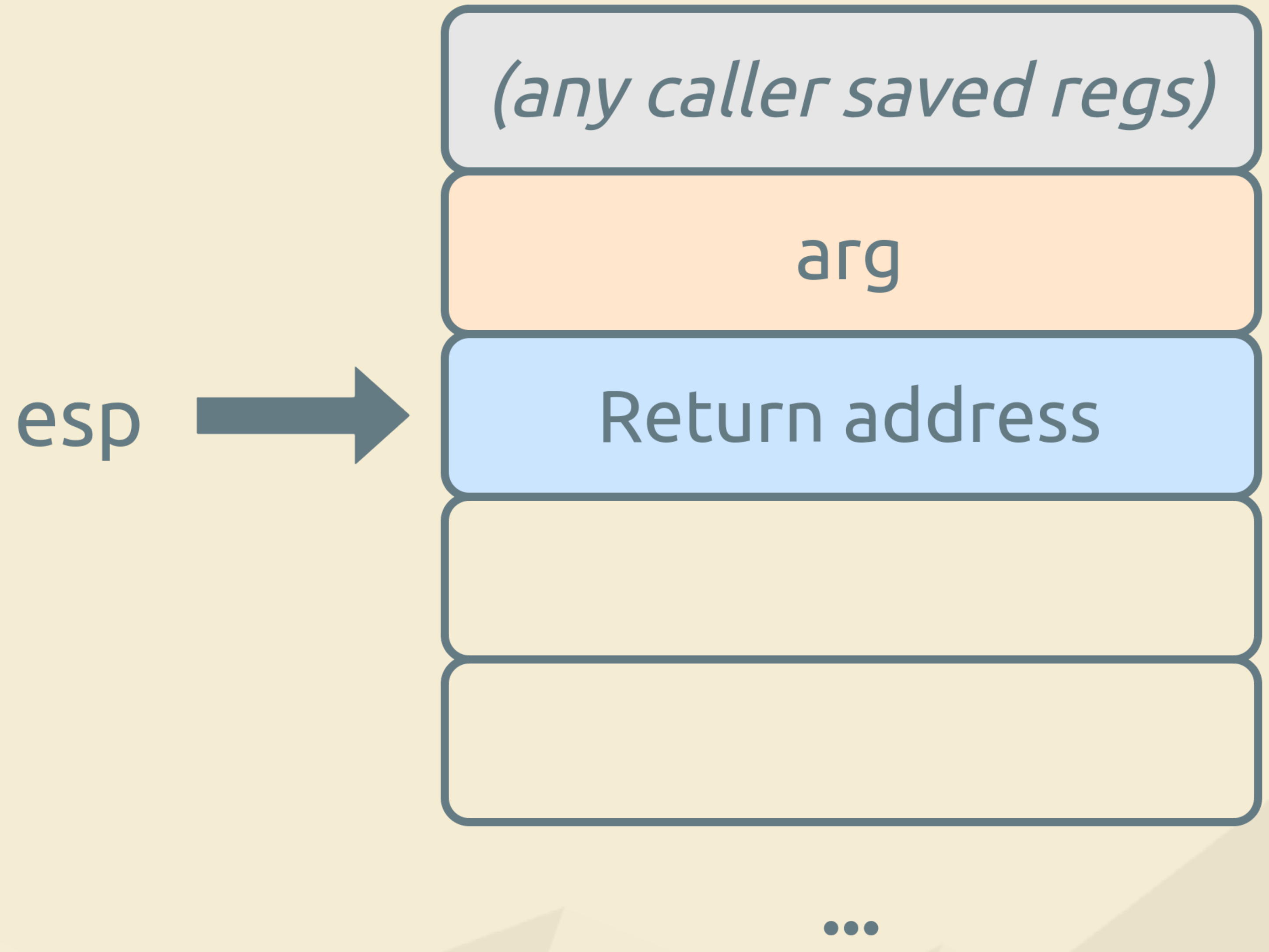
Stack when inside `func()`



# Single parameter

```
void func(T arg)
{
  // ...
}

func(t);
```

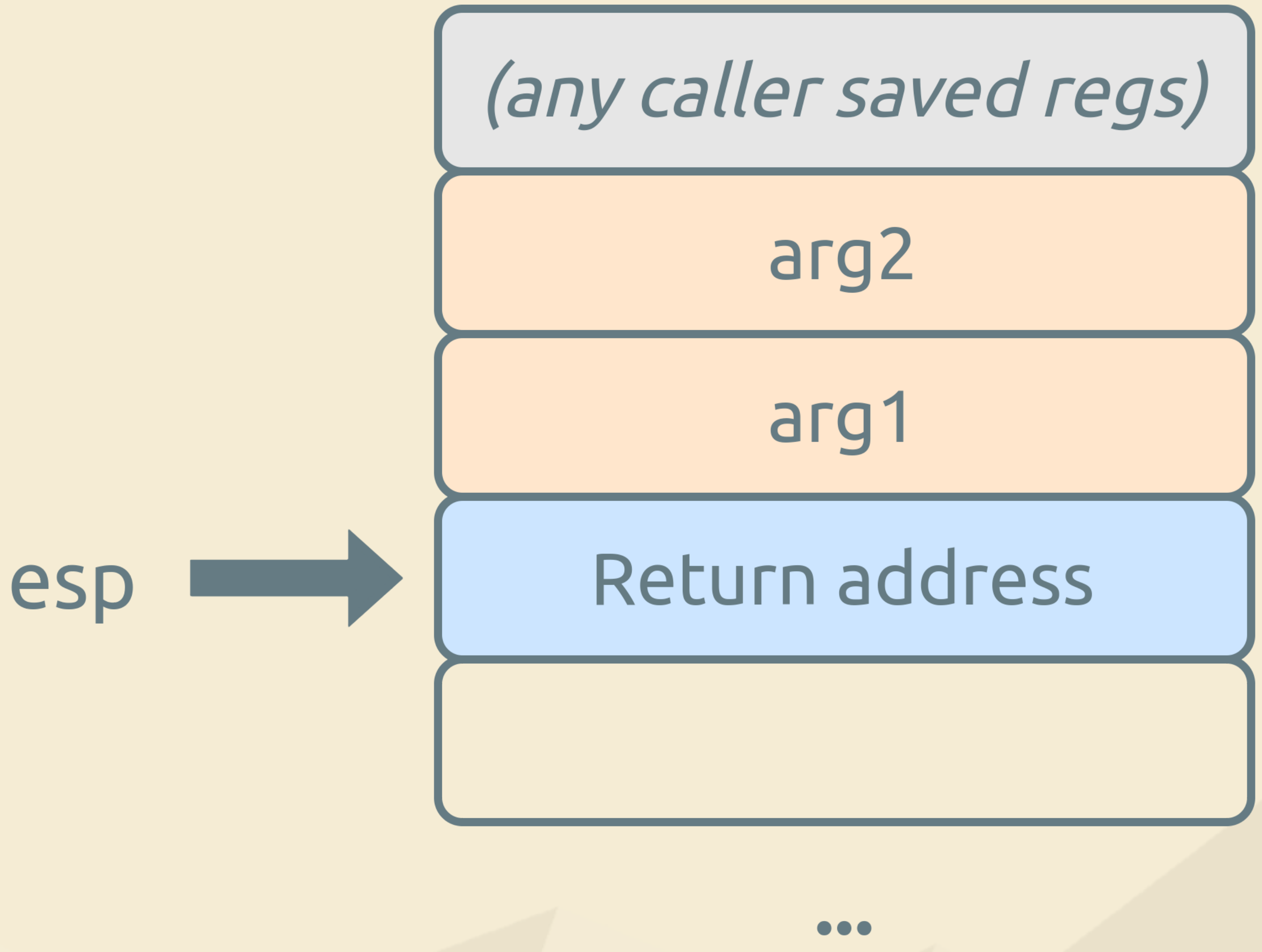
Stack when inside `func()`



# Multiple parameters

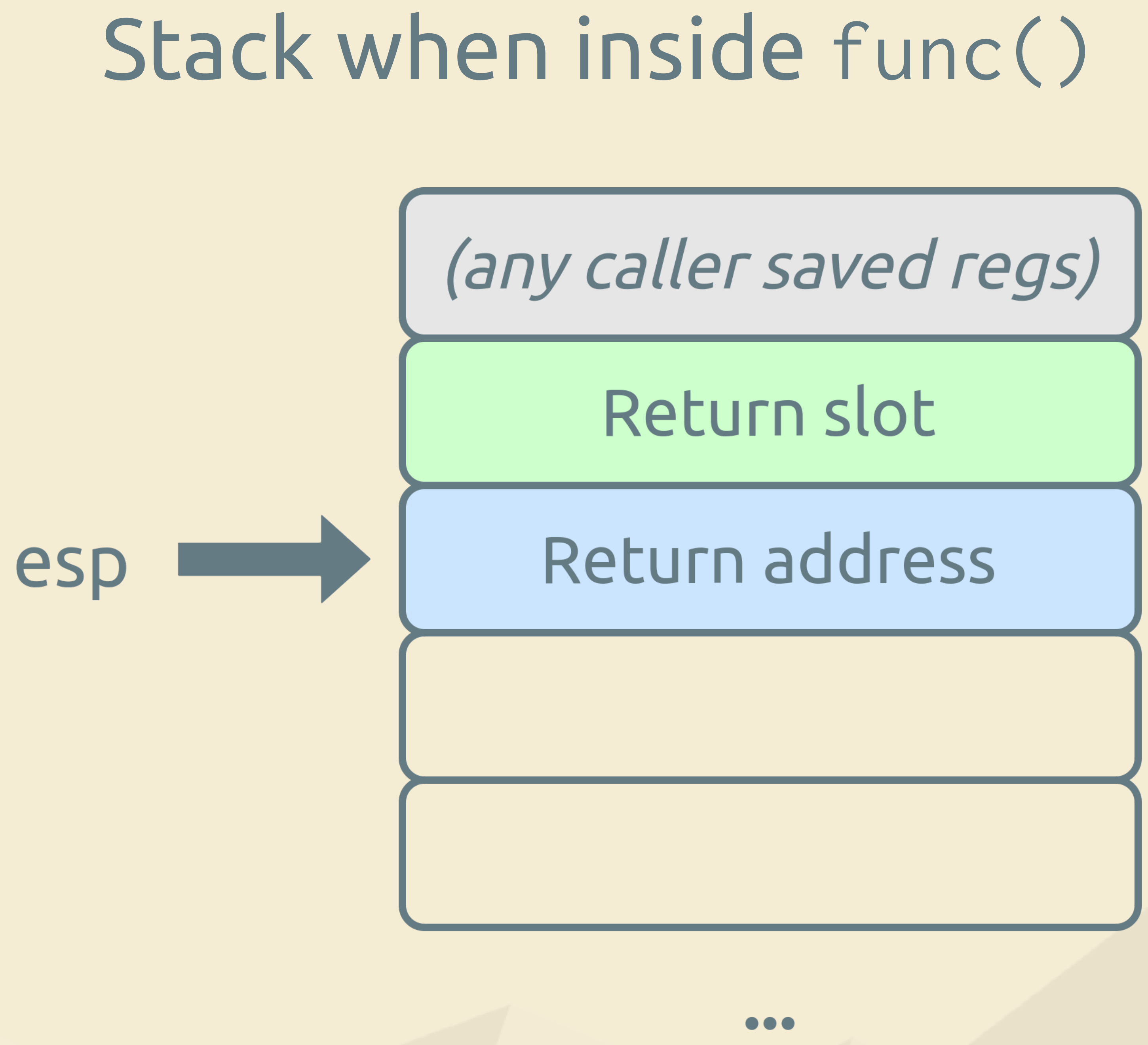
```
void func(T arg1, T arg2)
{
  // ...
}

func(t1, t2);
```

Stack when inside `func()`



# Return value

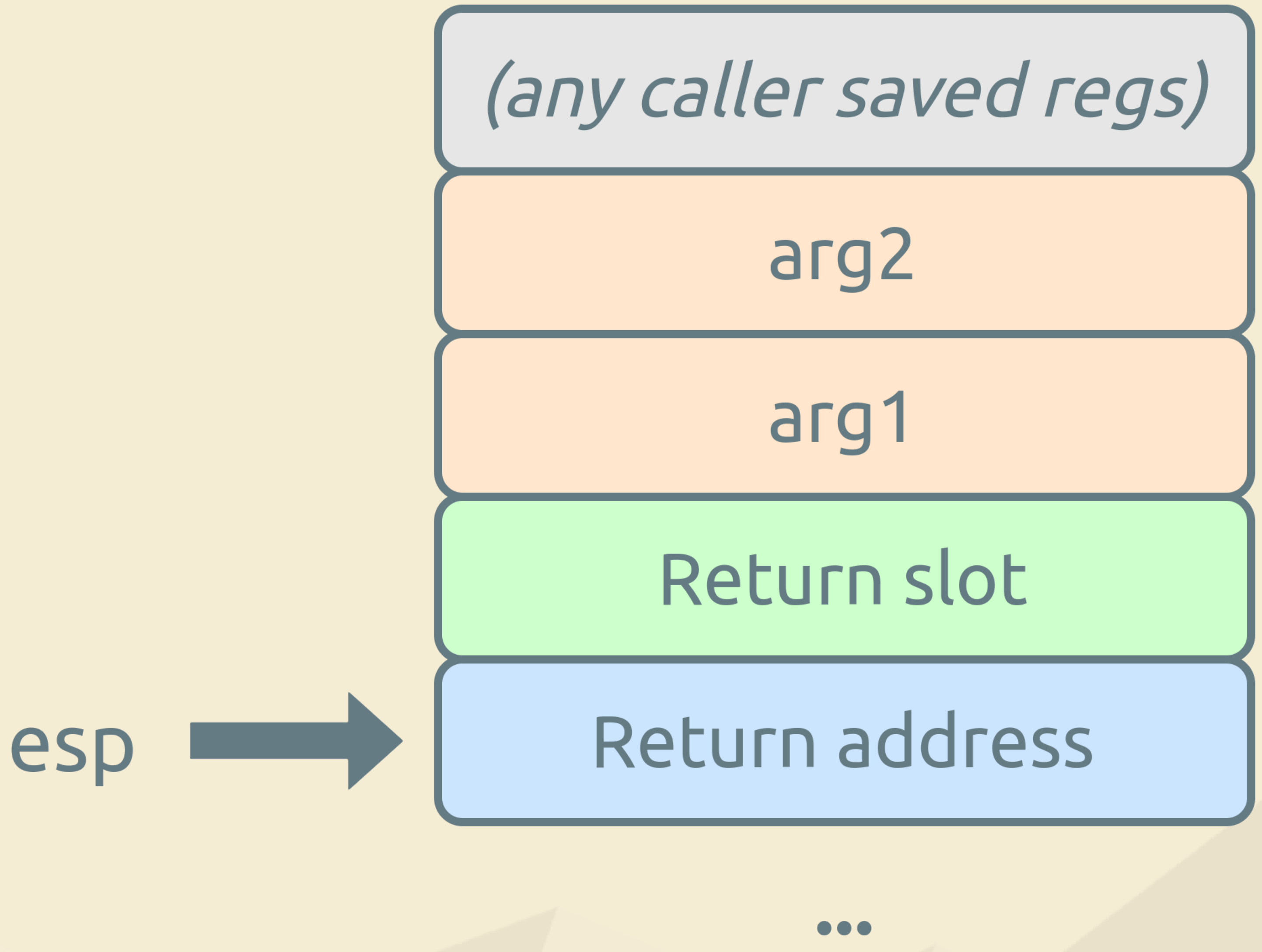
```
T func()
{
  // ...
}

T result = func();
```



# Parameters + return value combined

```
T func(T arg1, T arg2)
{
  // ...
}

T result = func(t1, t2);
```

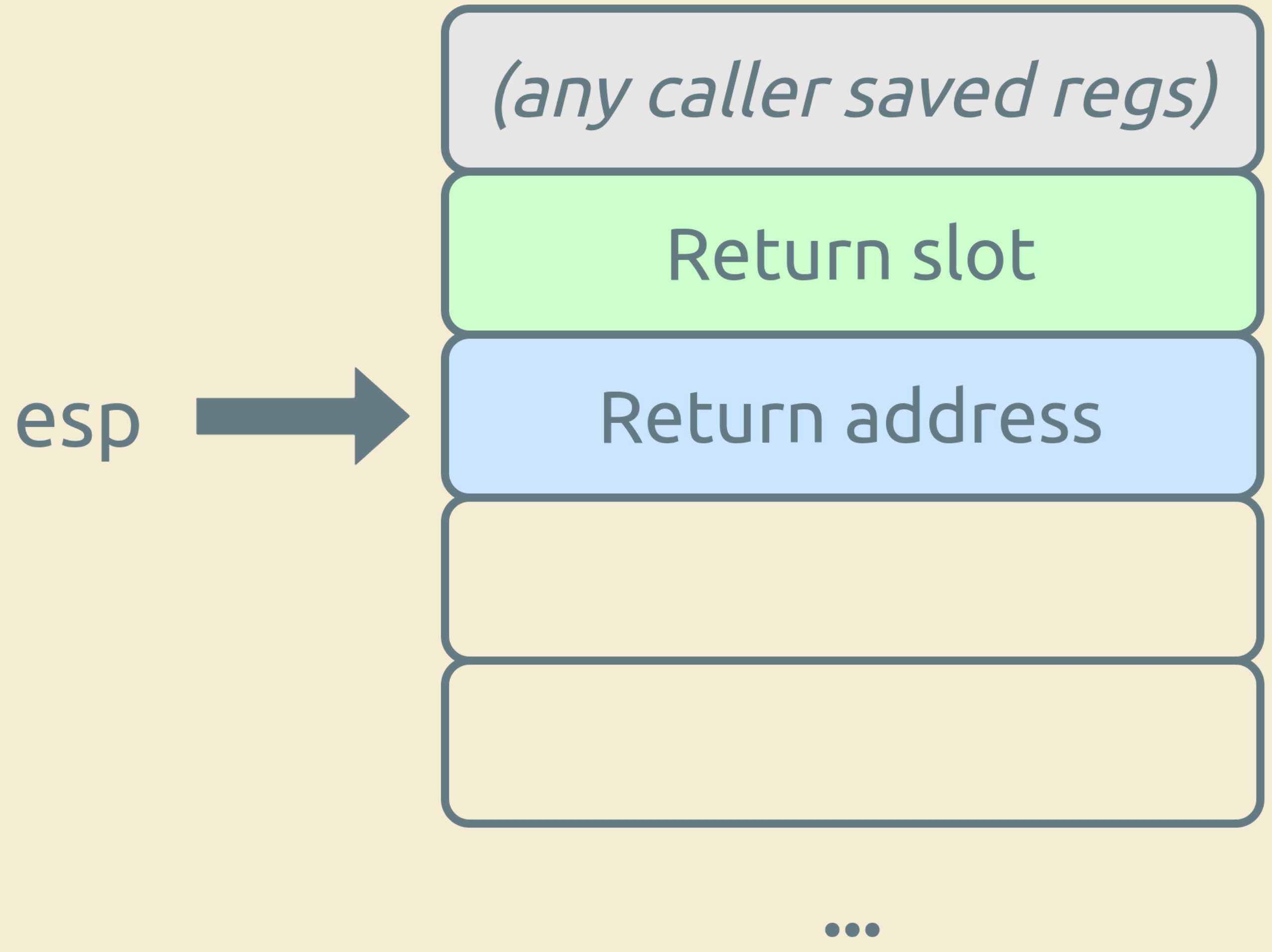
Stack when inside `func()`



# Back to NRVO

# NRVO is not always possible (1)

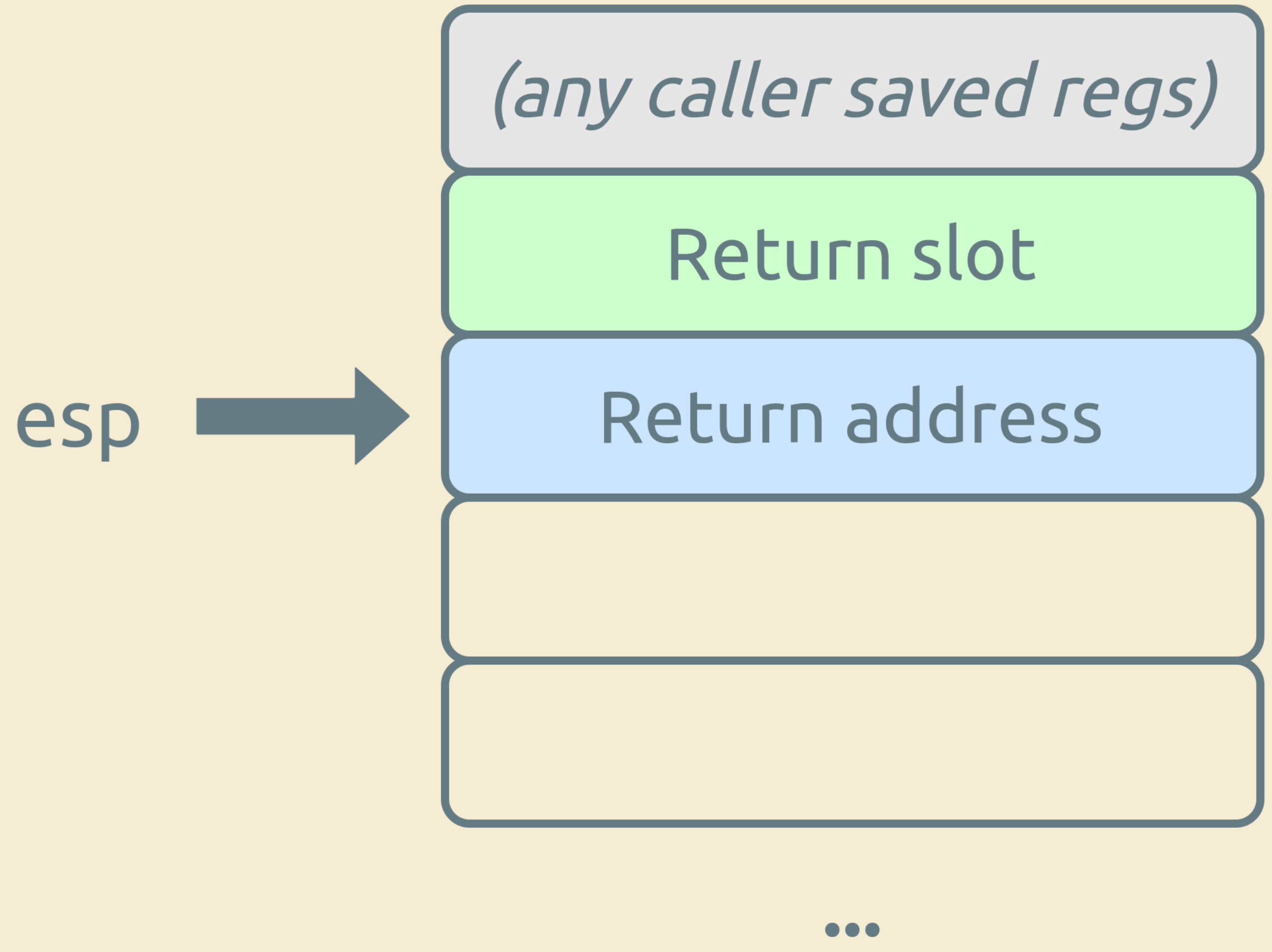
```
T func()
{
  T result;

  if (something)
    return {};   // prvalue

  return result; // lvalue
}

T x = func();
```




Not possible to allocate the return value into the return slot

# NRVO is not always possible (2)

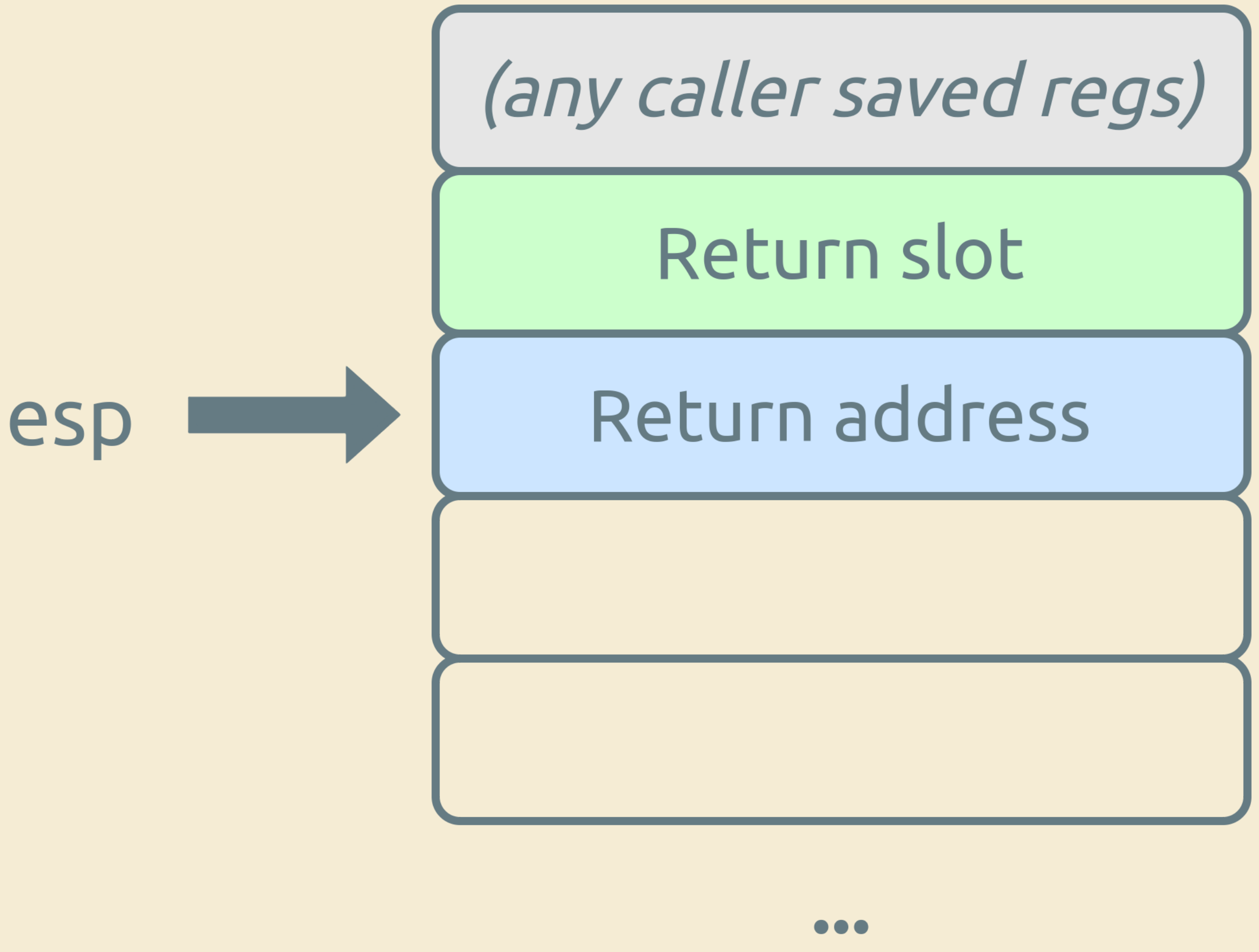
```
static T result;

T func()
{
  return result;
}

T x = func();
```




Returning an object of static storage duration

# NRVO is not always possible (3)

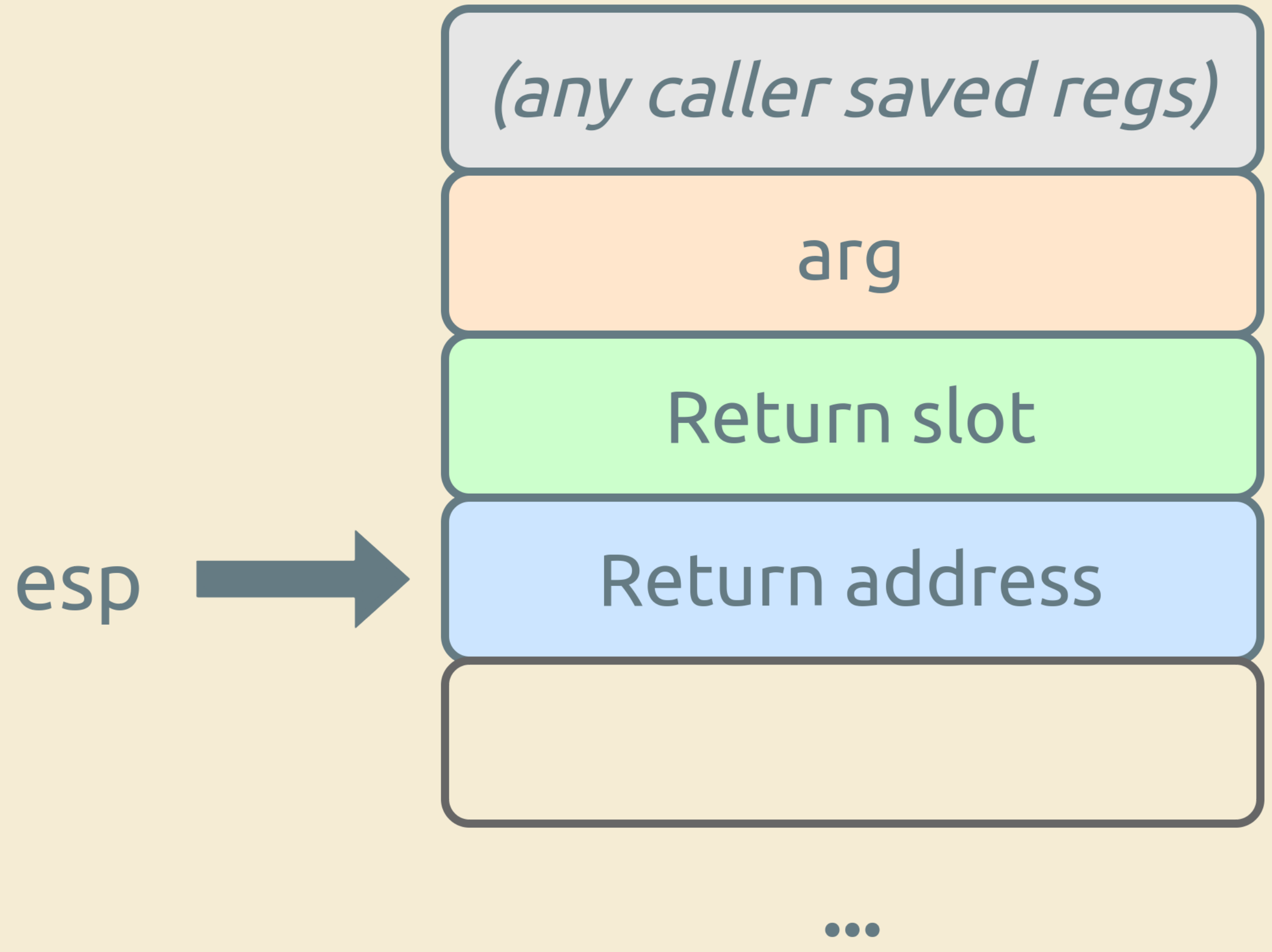
```cpp
struct U : T { /* Additional members */ };

T func()
{
  U result;

  return result;
}

T x = func();
```




Slicing 🍕

# NRVO is not always possible (4)

```
T func(T arg)
{
  return arg;
}

T x = func(T{});
```




Returning a function argument

# Summarizing RVO

> *RVO does not work when **there's no control over the physical location of the object to be elided.***

# Implicit move

## When even NRVO is not possible..

```cpp
T func(T arg)
{
  return arg;
}

T x = func(T{});
```

```
T()
T(&&)
~T()
~T()
```

Implicit **rvalue** conversion!

# Inadvertently disabling NRVO

## NRVO

```
T func()
{
  T result;

  return result;
}

T x = func();
```

## No NRVO

```
T func()
{
  T result;

  return std::move(result);
}

T x = func();
```

Don’t try to be too clever.. 🤓

# Guidelines

- Don’t be afraid to return a non-POD type by value,
- Don’t be too smart, let the compiler do the work for you,
- Implement your move constructor/`operator=`,
- Use compile-time programming if possible,
- Keep your functions short.

# Quiz revisited

```cpp
struct Number {
  int value_ = {};
};

class T {
public:
  T(const Number &n) : n_{n} {}

  T(const T &) { puts("Copy c'tor"); }

  Number get() { return n_; }

private:
  Number n_;
};
```

```cpp
static T create(Number &&n) {
  return T{std::move(n)};
}

int main() {
  T x = T{create(Number{42})};

  return x.get().value_;
}
```

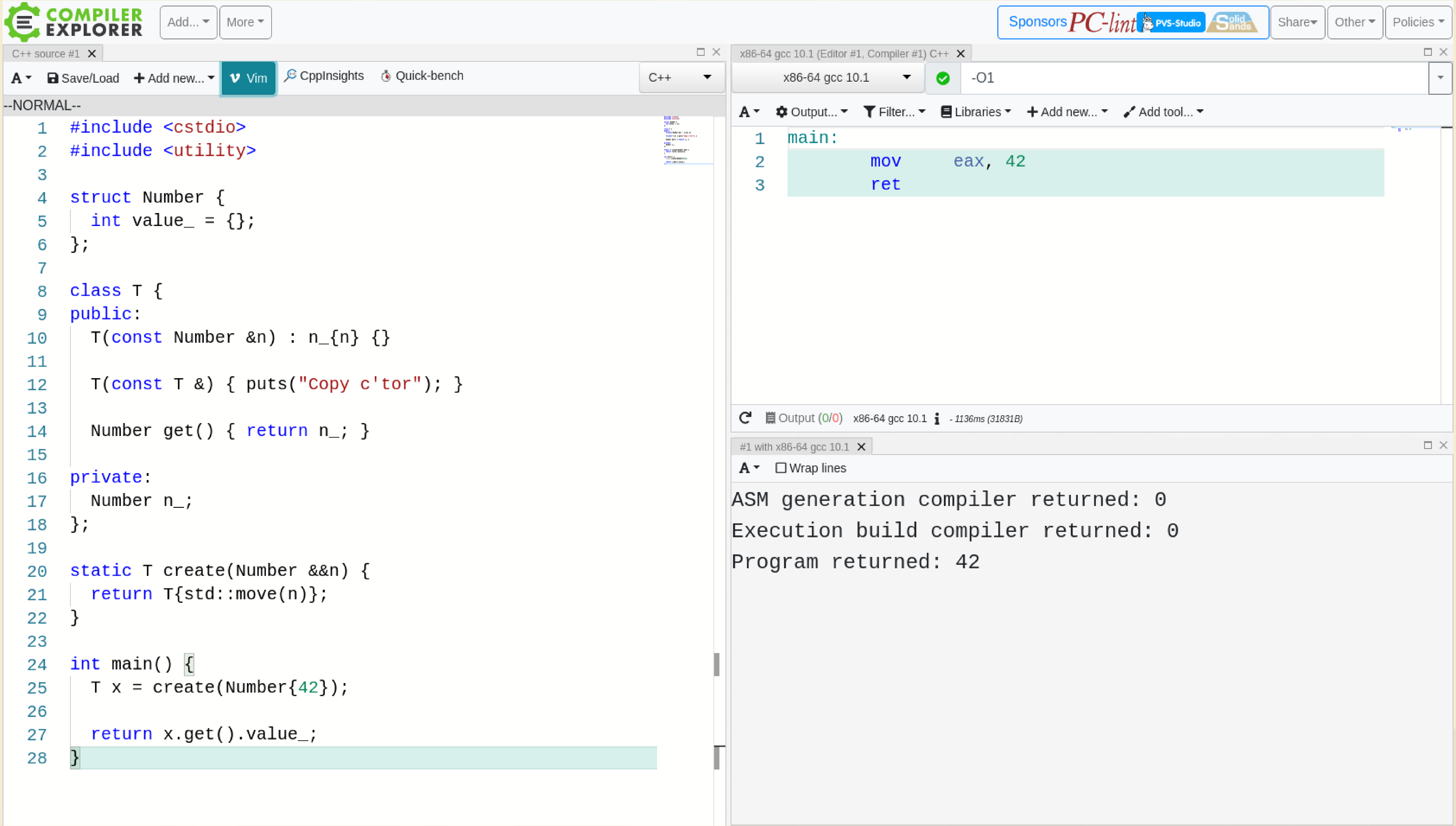
What’s the output?

# Quiz revisited

```cpp
struct Number {
  int value_ = {};
};

class T {
public:
  T(const Number &n) : n_{n} {}

  Number get() { return n_; }

private:
  Number n_;
};
```

```cpp
int main() {
  T x = T{Number{42}};

  return x.get().value_;
}
```

What’s the output?


COMPILER EXPLORER
Add...
More
Sponsors PC-lint PVS-Studio Solid Sands
Share
Other
Policies
C++ source #1
Save/Load
Add new...
Vim
CppInsights
Quick-bench
C++
--NORMAL--
#include <cstdio>
#include <utility>

struct Number {
  int value_ = {};
};

class T {
public:
  T(const Number &n) : n_{n} {}

  T(const T &) { puts("Copy c'tor"); }

  Number get() { return n_; }

private:
  Number n_;
};

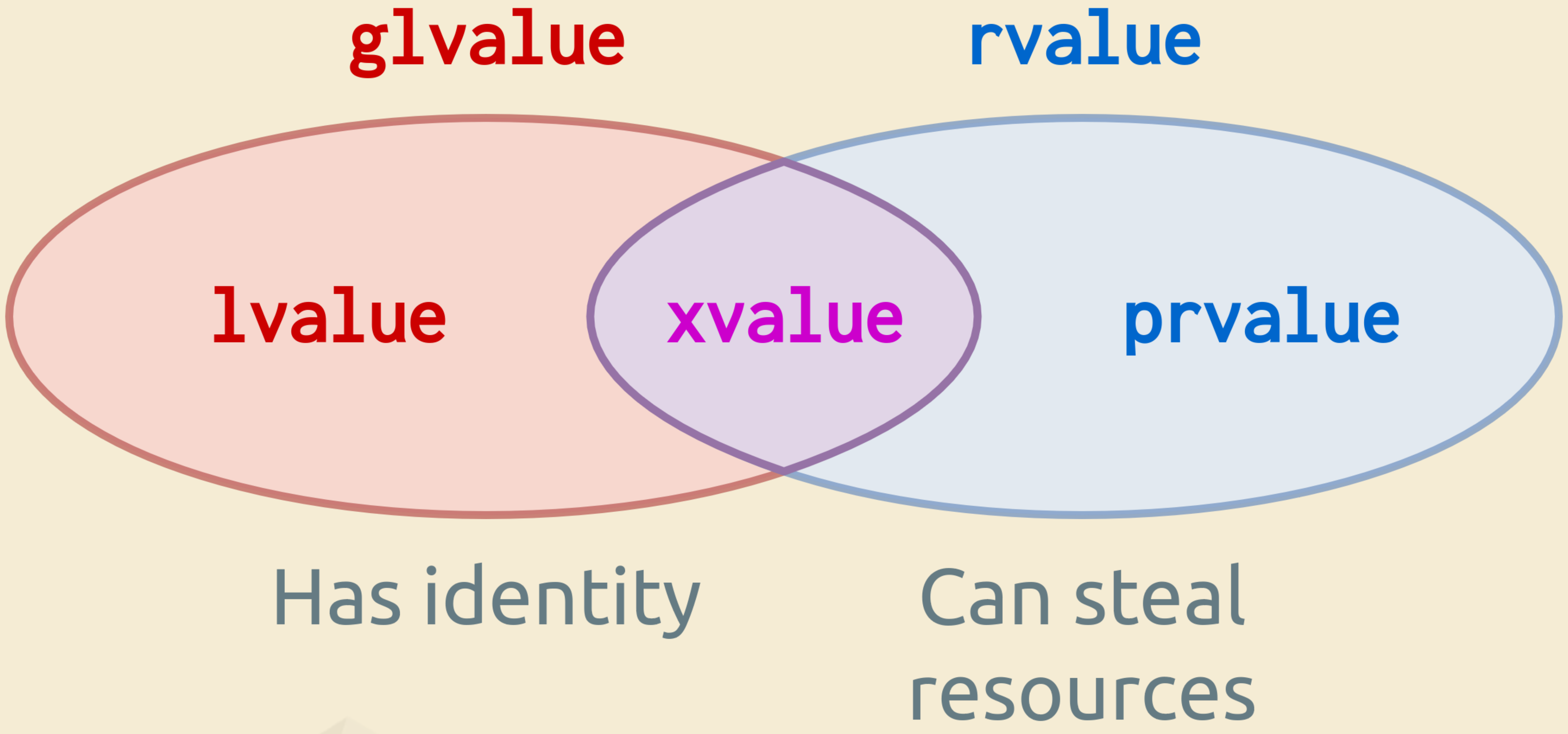
static T create(Number &&n) {
  return T{std::move(n)};
}

int main() {
  T x = create(Number{42});

  return x.get().value_;
}
x86-64 gcc 10.1 (Editor #1, Compiler #1) C++
x86-64 gcc 10.1
-O1
Output...
Filter...
Libraries
Add new...
Add tool...
main:
        mov     eax, 42
        ret
Output (0/0)
x86-64 gcc 10.1
- 1136ms (31831B)
#1 with x86-64 gcc 10.1
Wrap lines
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 42

# Conclusions

# C++ value categories



# Copy/move elision (1)

- Copy elision: part of the standard,
- Temporary materialization: part of the standard,
- URVO and NRVO: unofficial terms,
- Copy elision: allows, does not guarantee,
- Temporary materialization: mandates.

# Copy/move elision (2)

- Temporary materialization: `prvalue` to `xvalue` conversion,
- `prvalue`s are **not** moved from,
- Implicit move: a RVO that happens even without copy elision.

# End

Thank you 😃