The value categories

The biggest change since C++11.

scc

TEAMT5

杜 浦 數 位 安 全

Persistent Cyber Threat Hunters

The lvalue and the rvalue

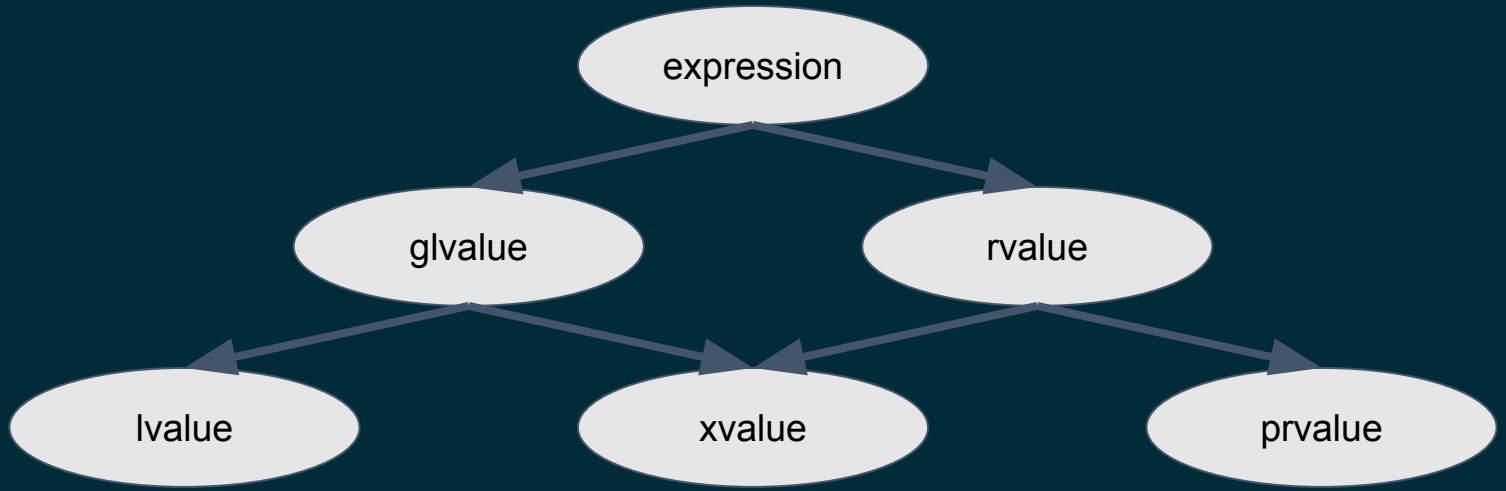
TEAMT5
杜浦數位安全

In the C programming language:

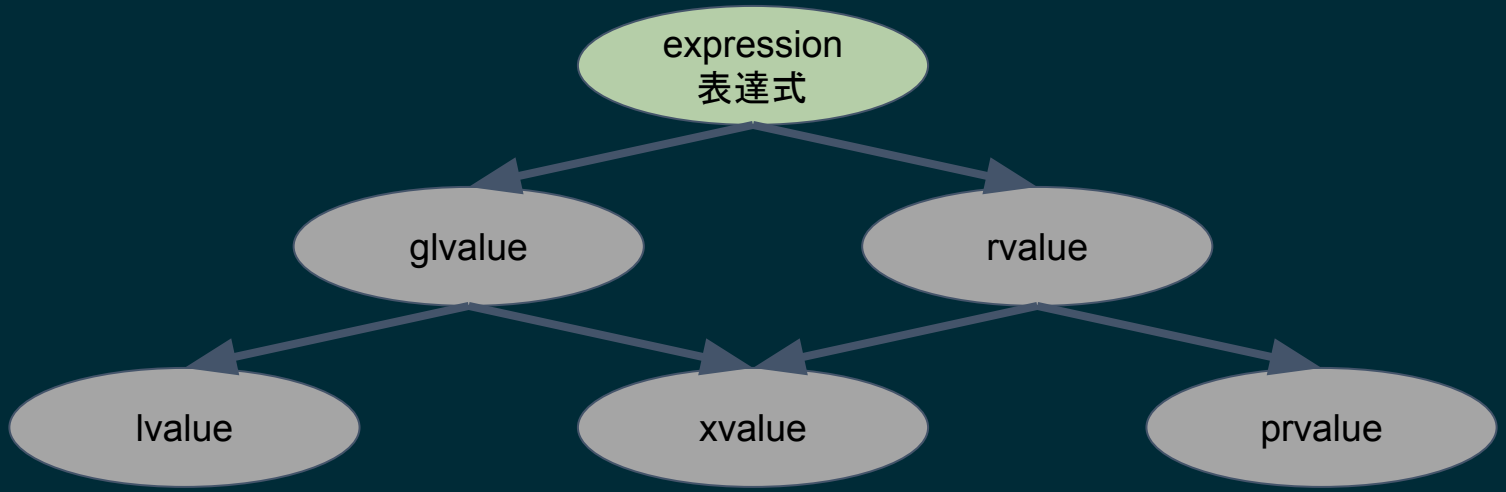
An *Object* is a named **region of storage**;
an lvalue is an expression referring to an object.

The C programming language Kernighan, Brian; Ritchie, Dennis M.

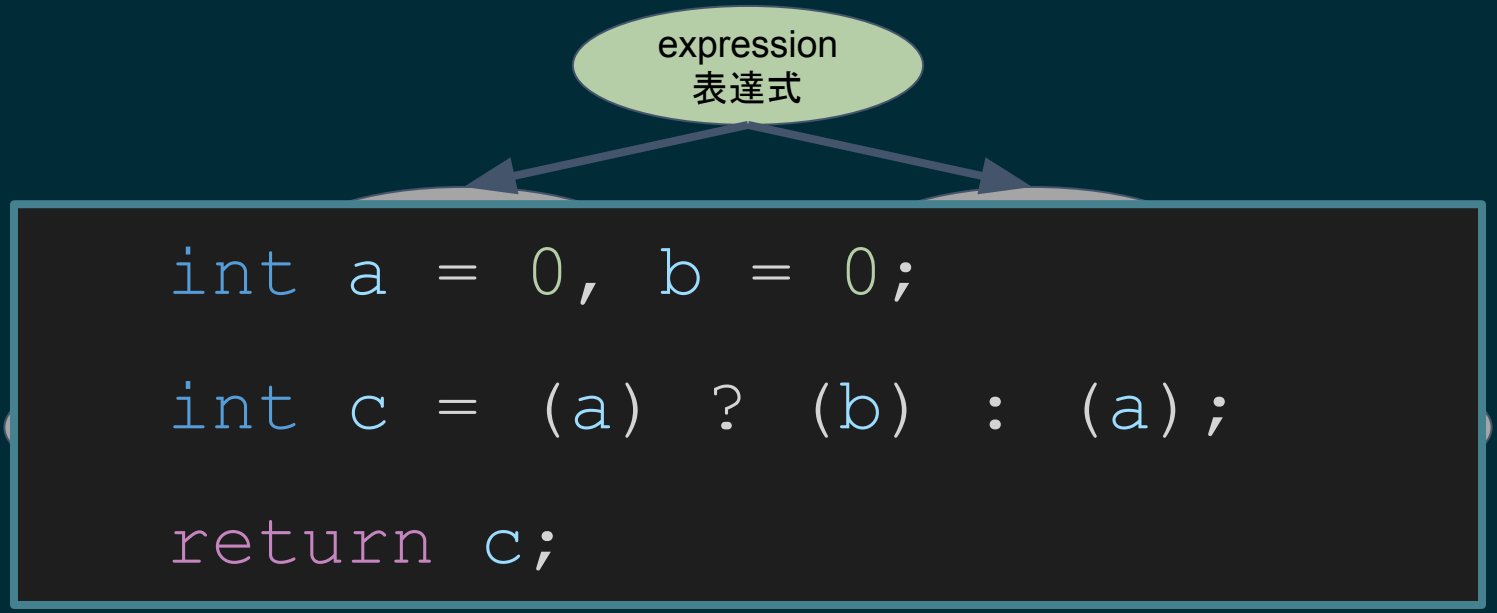
Value categories after C++11



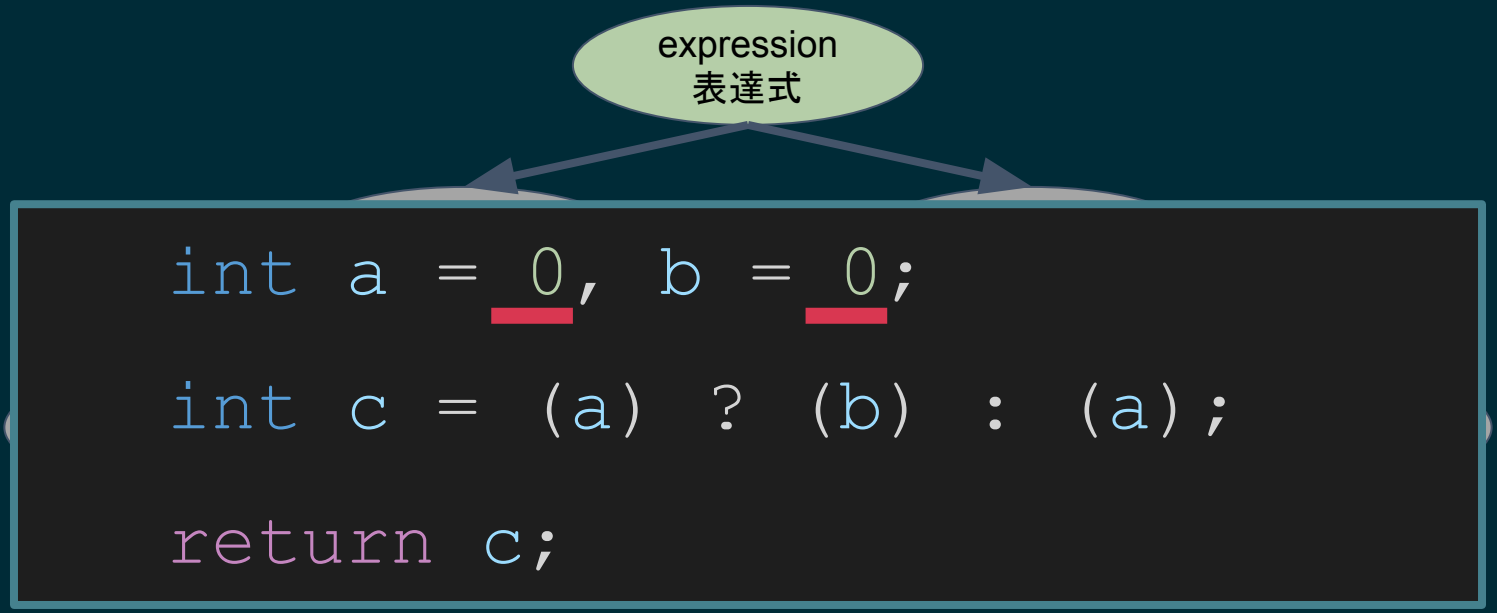
Value categories after C++11



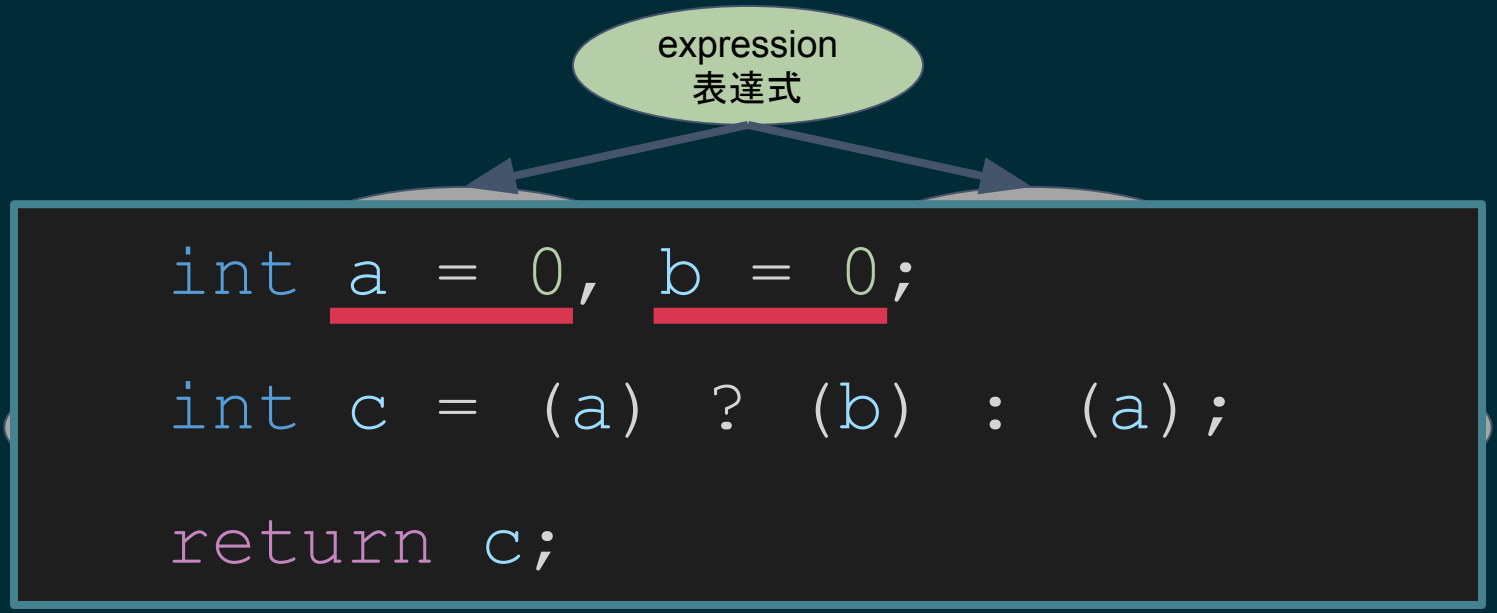
Value categories after C++11



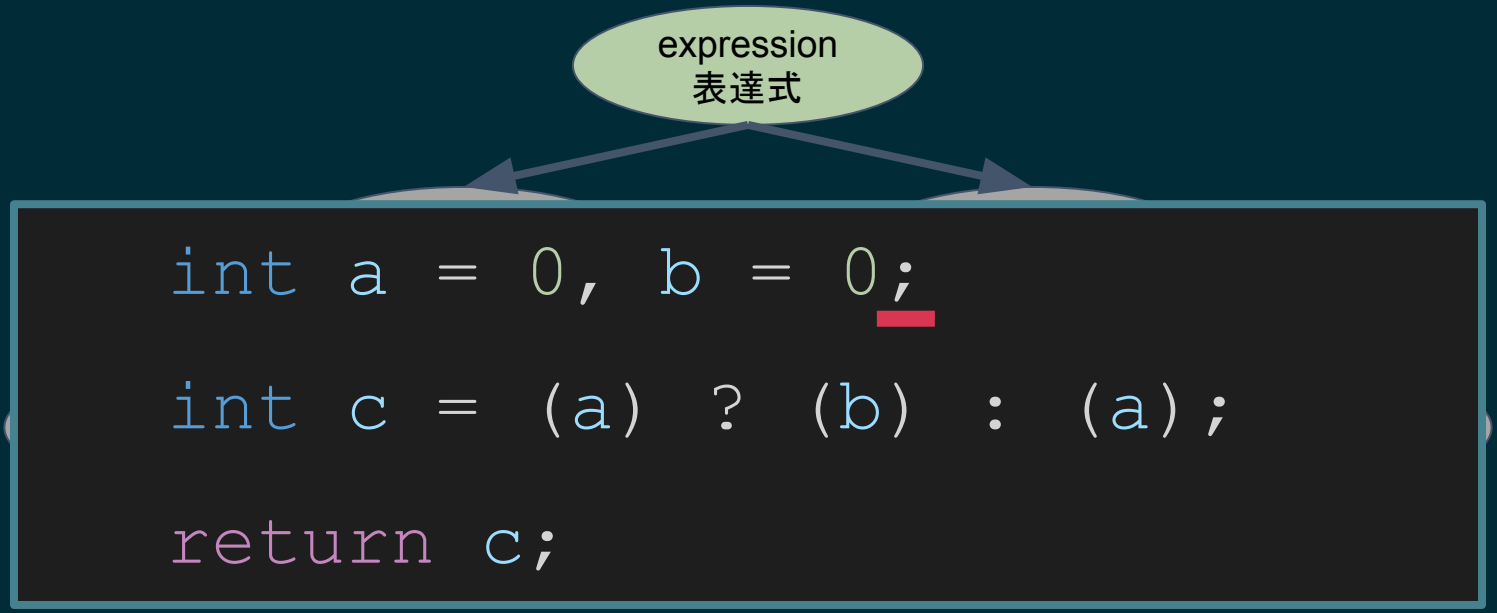
Value categories after C++11



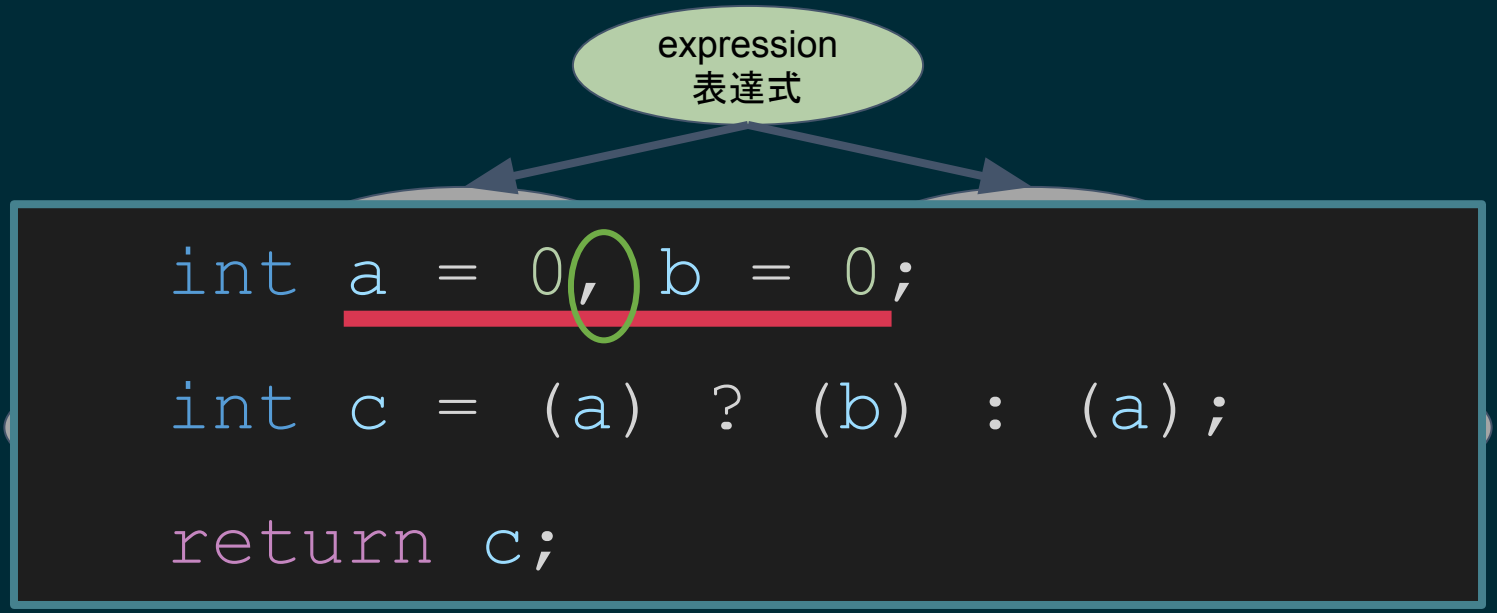
Value categories after C++11



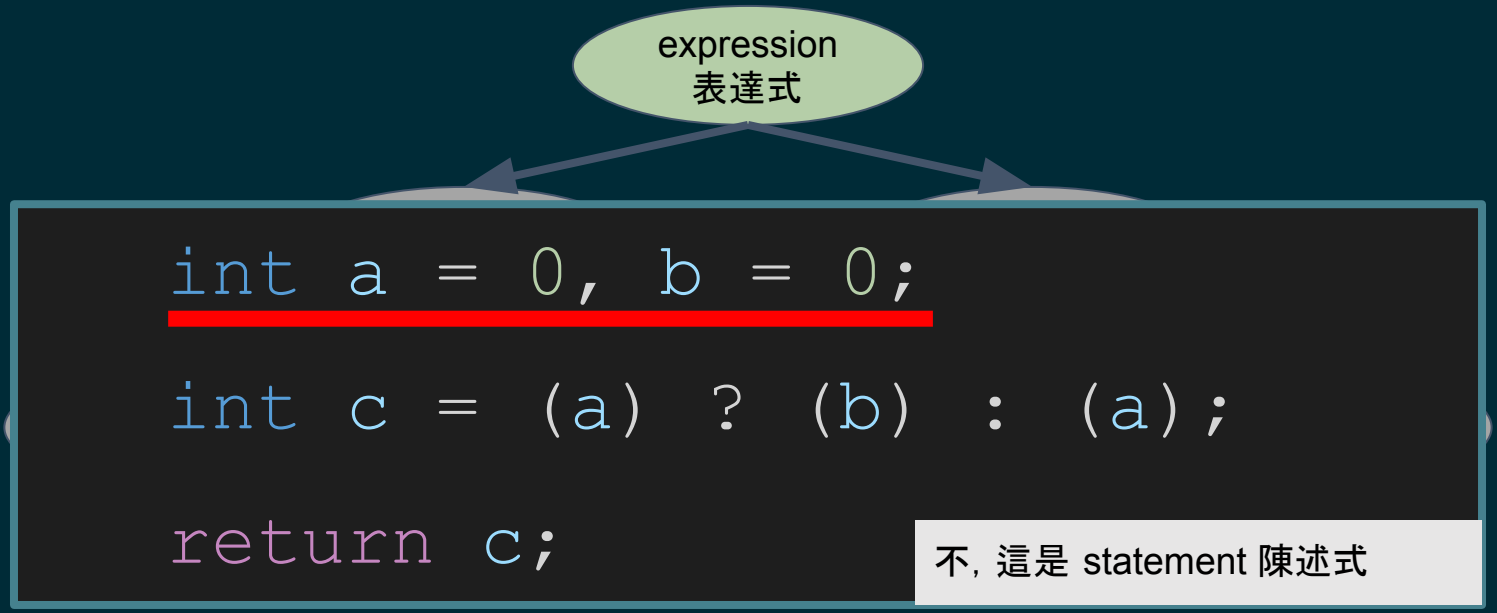
Value categories after C++11



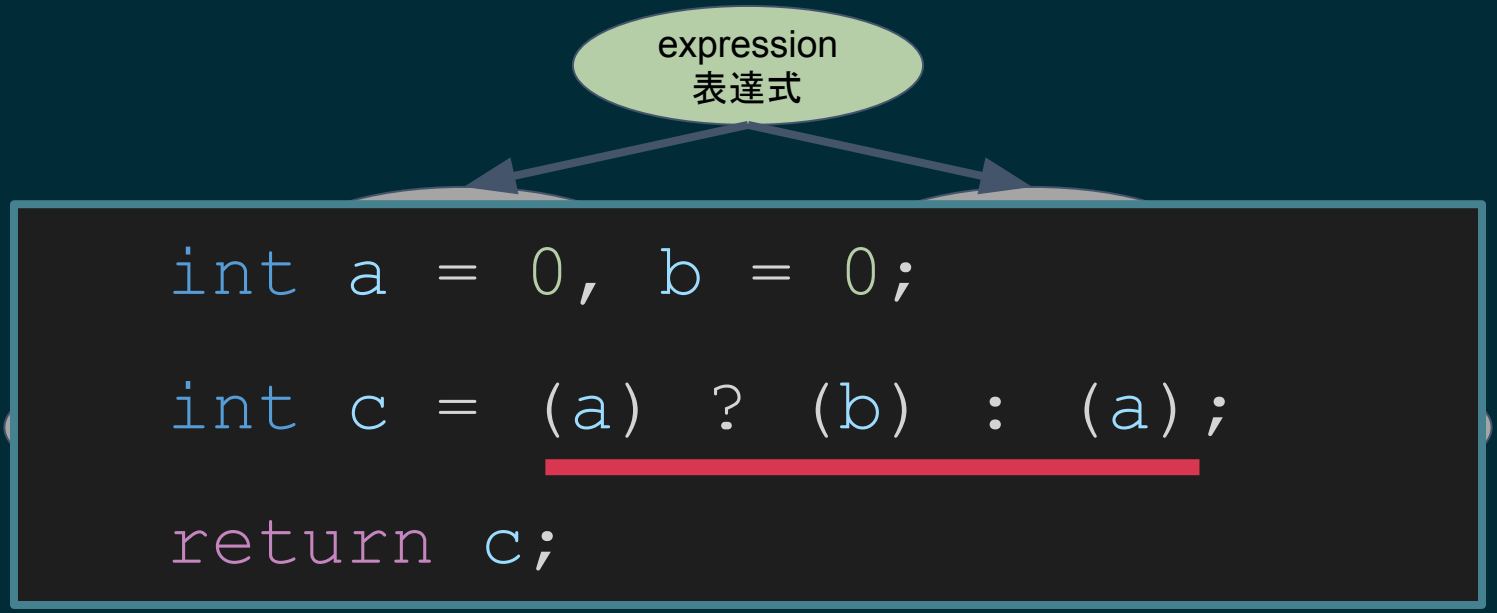
Value categories after C++11



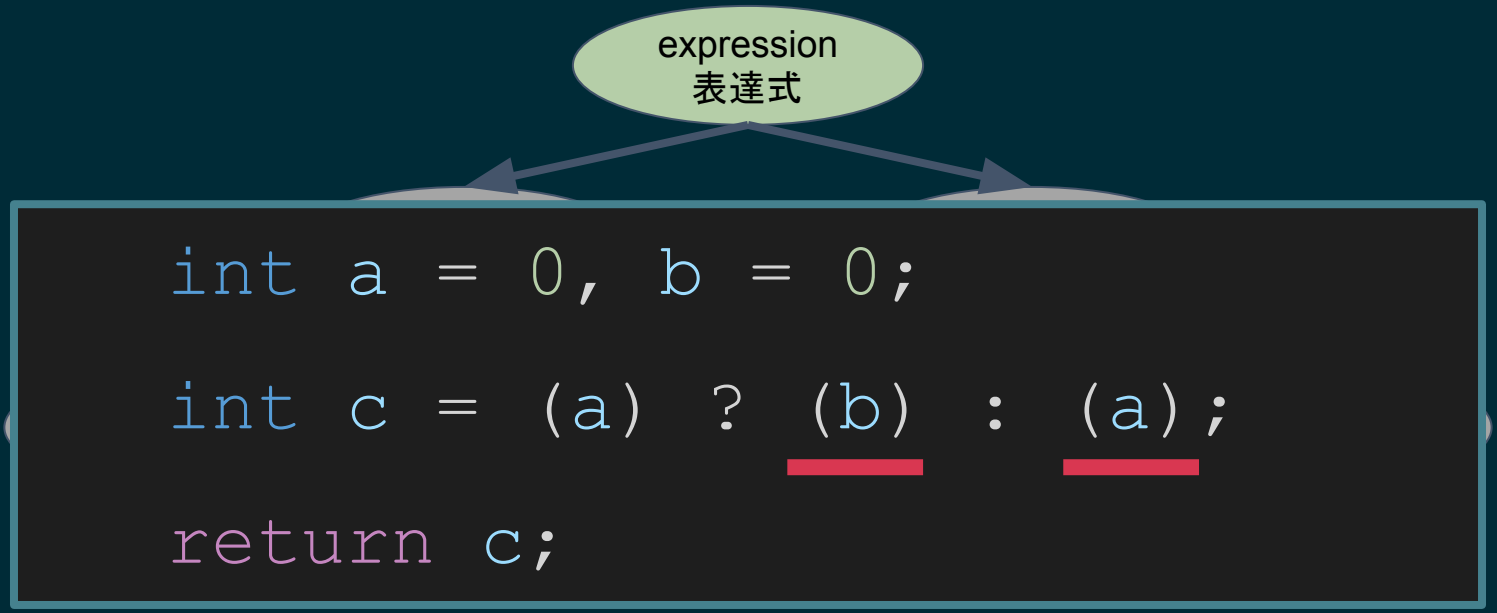
Value categories after C++11



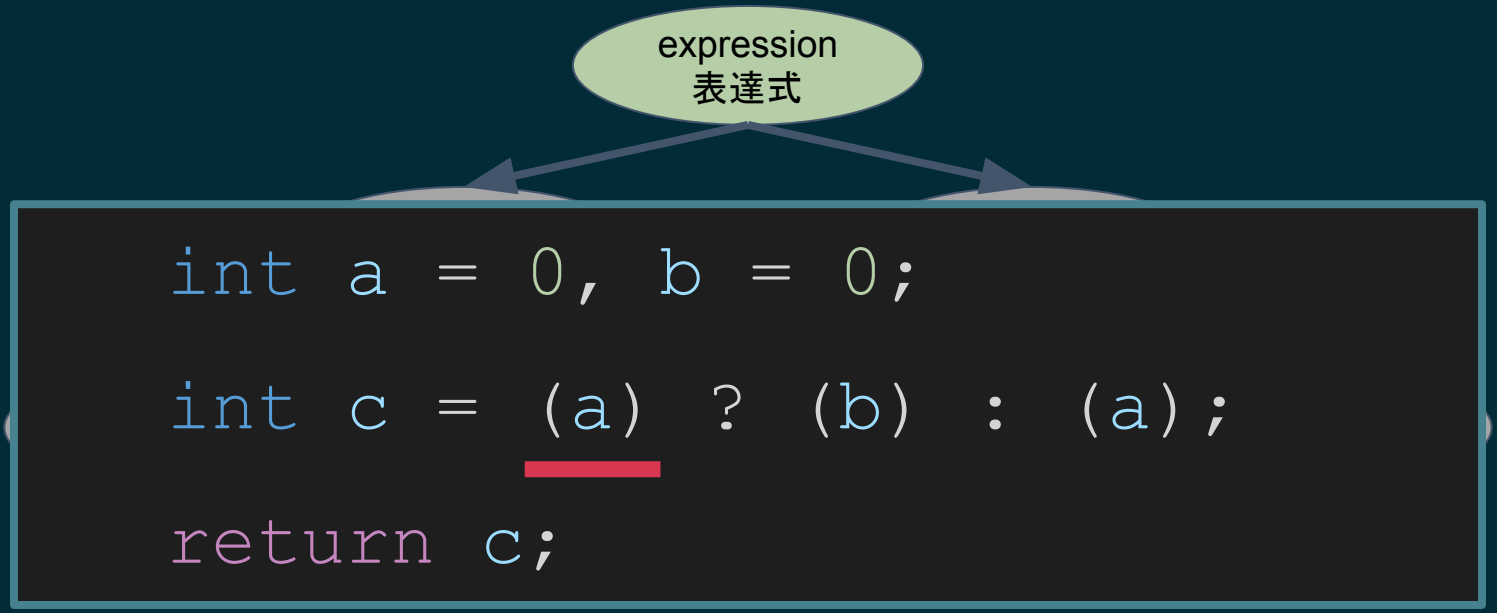
Value categories after C++11



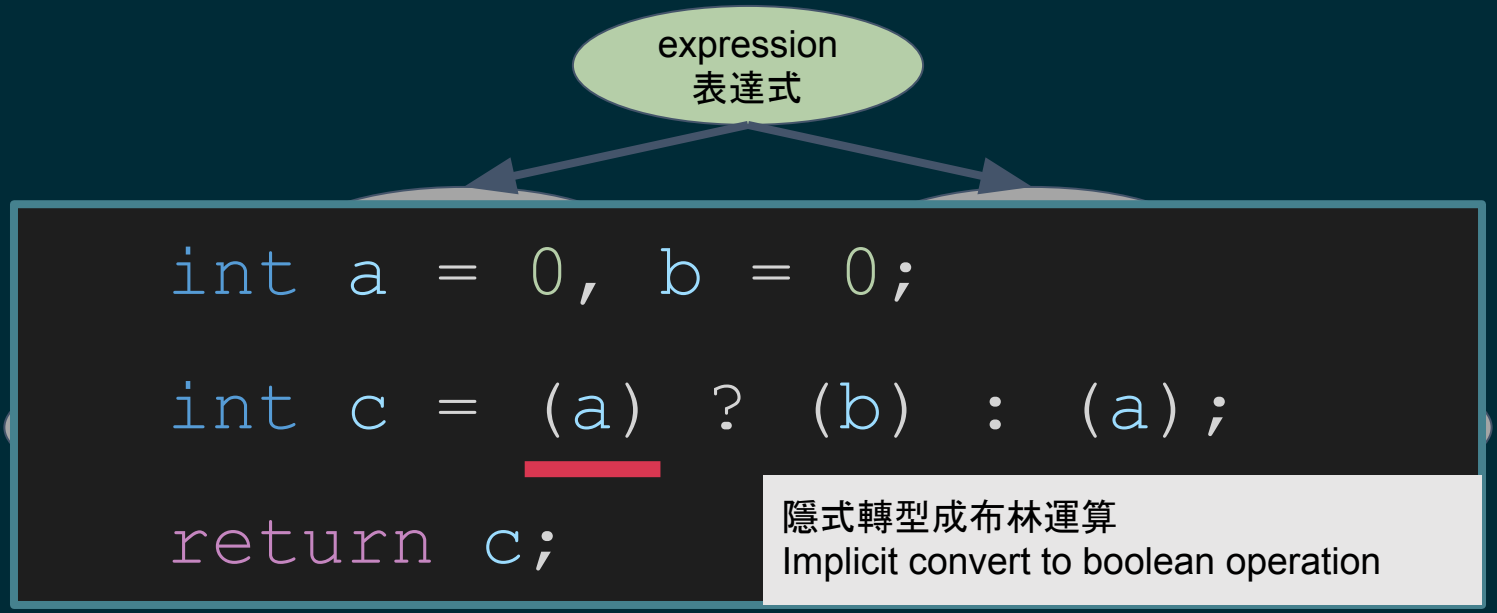
Value categories after C++11



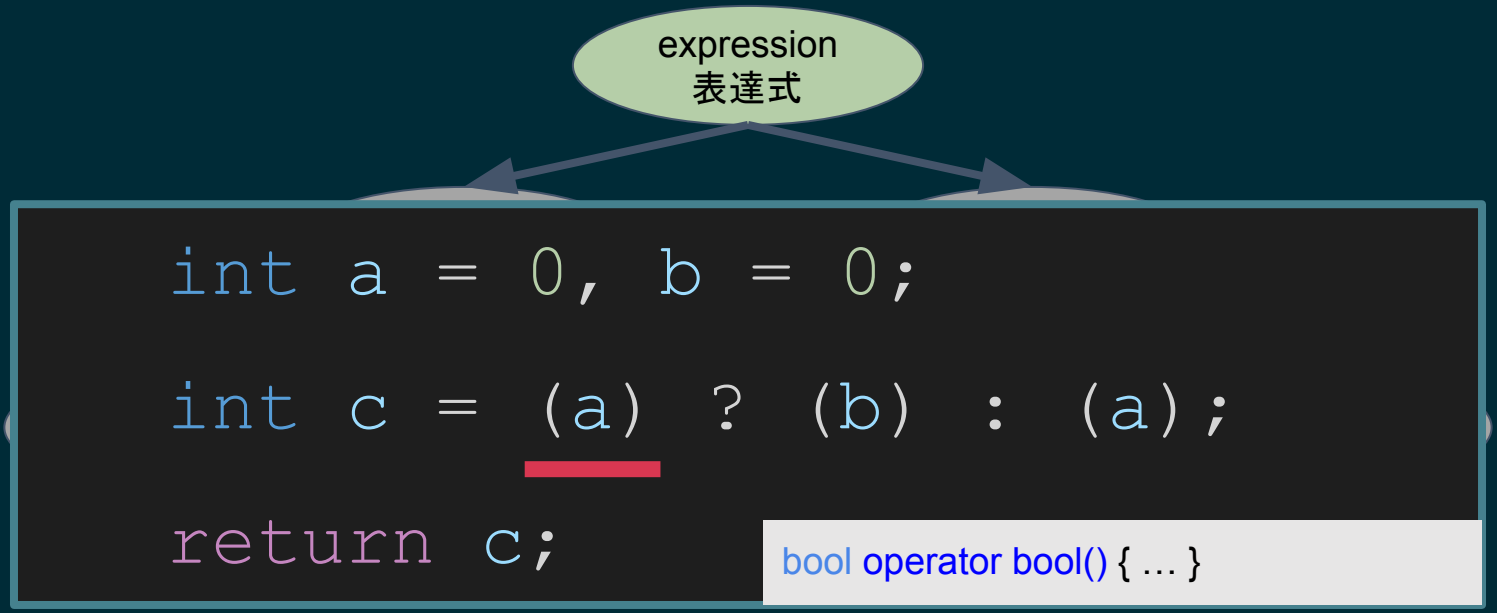
Value categories after C++11



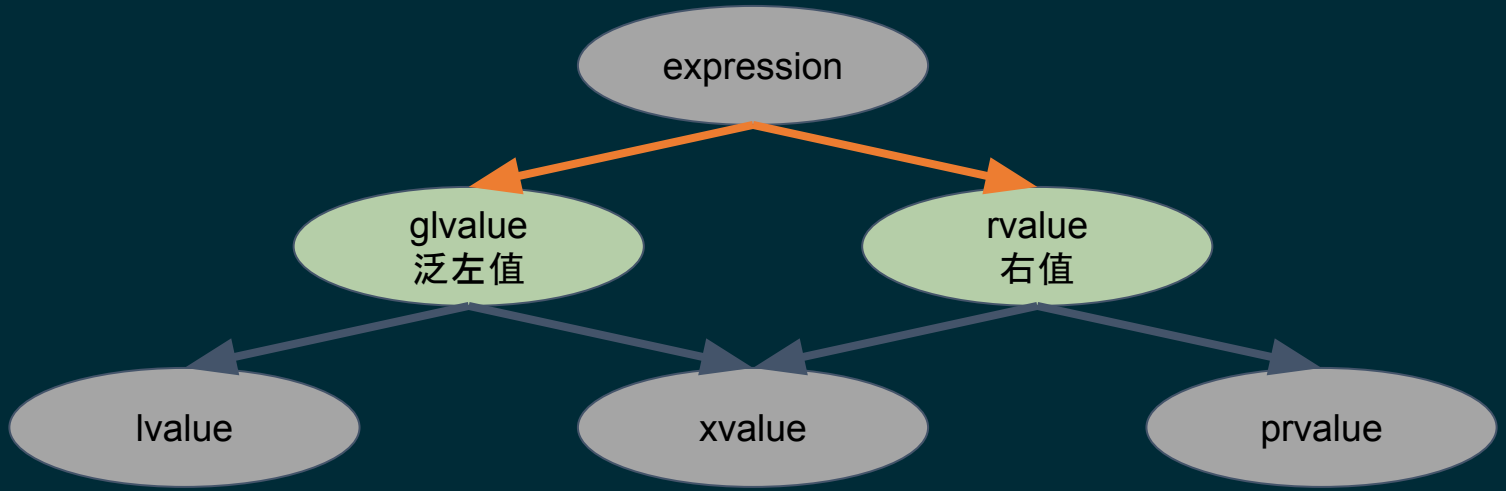
Value categories after C++11



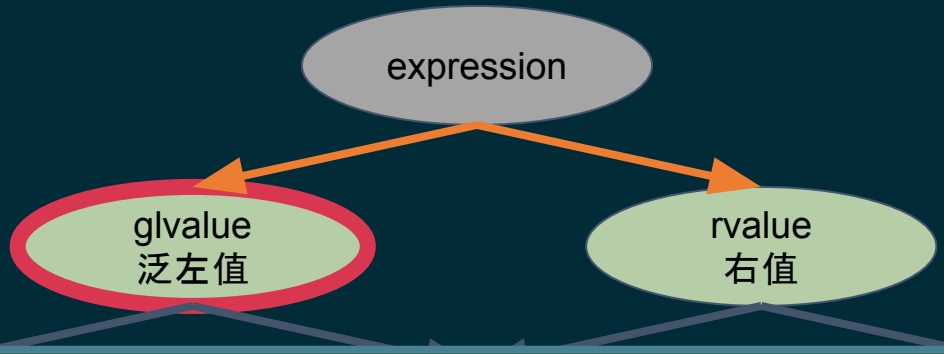
Value categories after C++11



Value categories after C++11



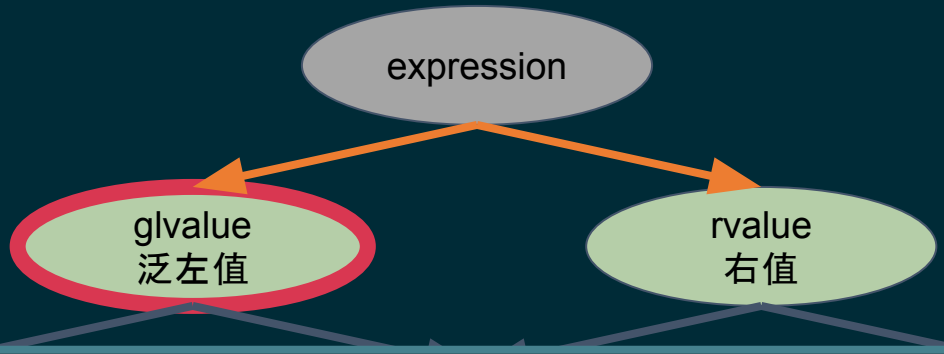
Value categories after C++11




```
int a = 0, b = 0;

int c = (a) ? (b) : (a);

return c;
```

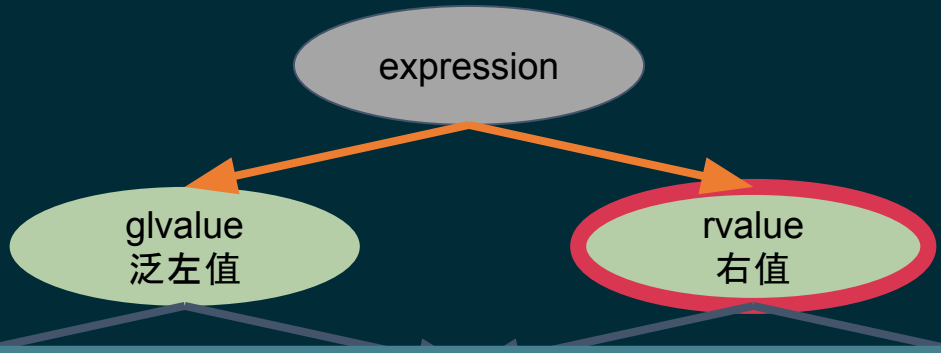
Value categories after C++11

TEAMT5
杜浦數位安全



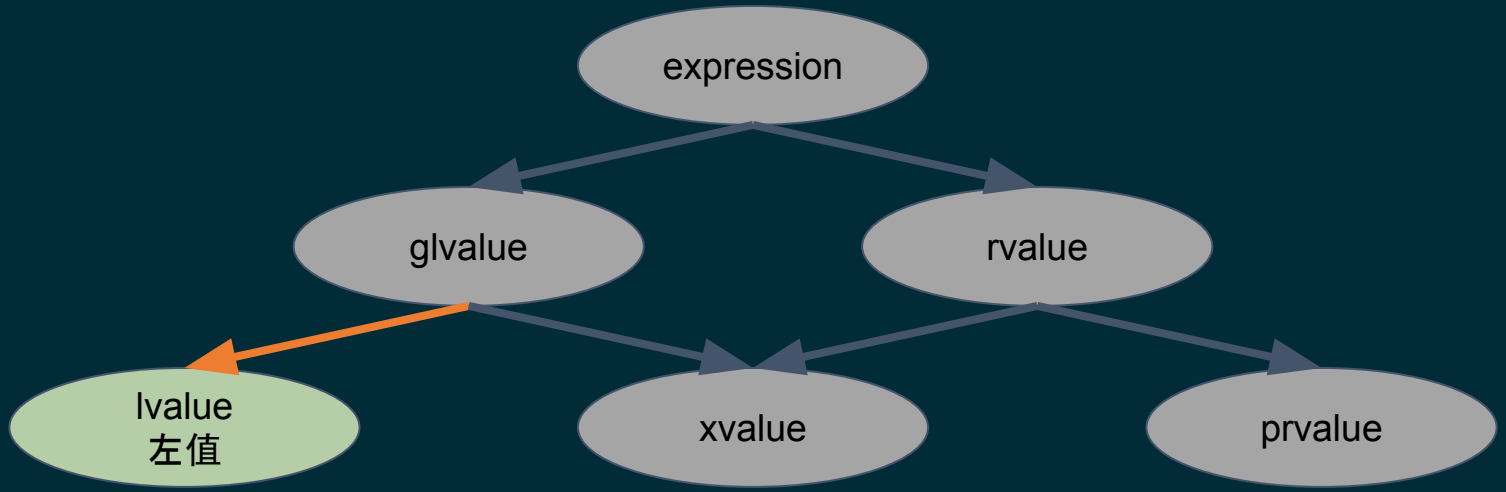

```
int a = 0, b = 0;

int c = (a) ? (b) : (a);

return c;
```

Value categories after C++11




```
int a = 0, b = 0;
int c = (a) ? (b) : (a);
return c;
```

Value categories after C++11



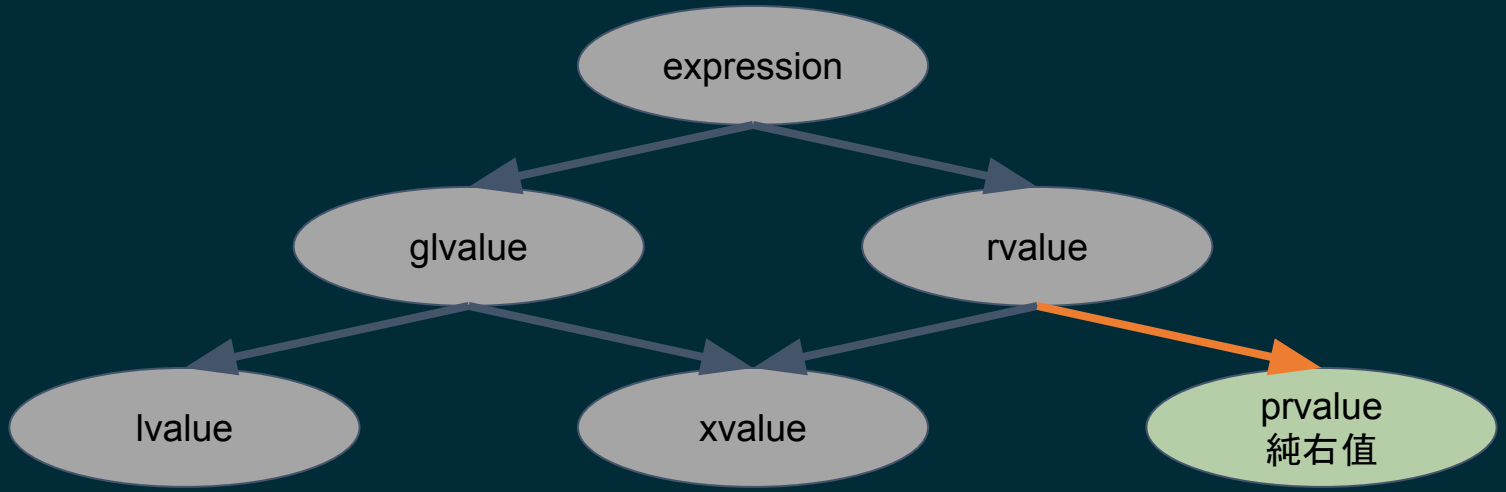
Value categories after C++11

TEAMT5
杜浦數位安全

```
int a = 0, b = 0;

int c = (a) ? (b) : (a);

return c;
```

Value categories after C++11

TEAMT5
杜浦數位安全



Value categories after C++11

TEAMT5 杜浦數位安全

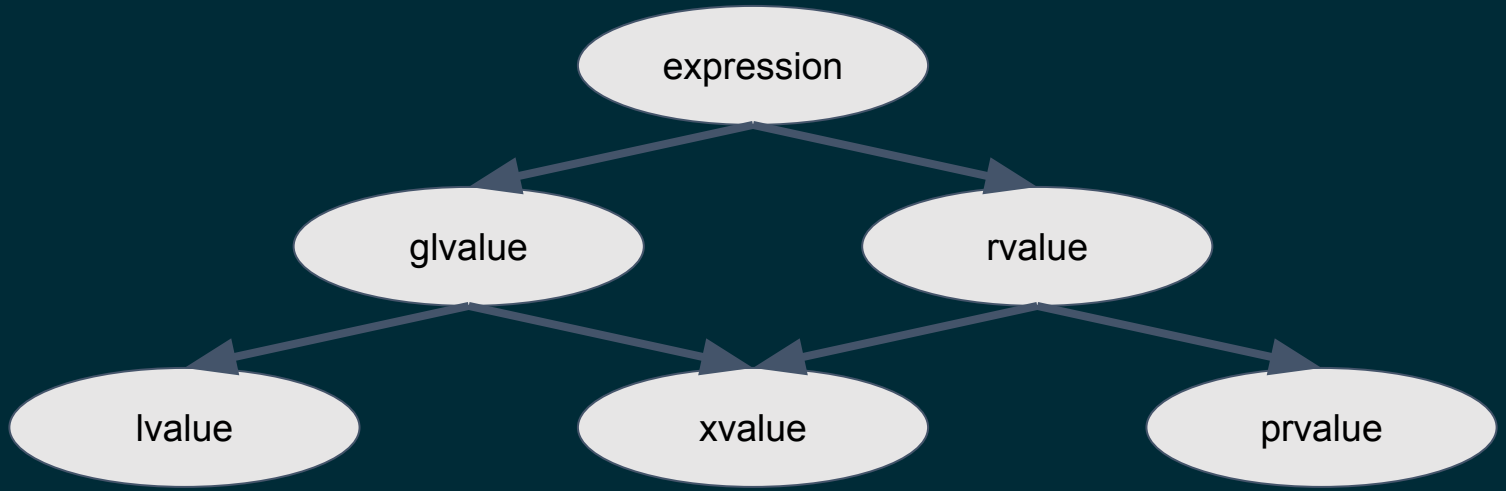
```
int a = 0, b = 0;

int c = (a) ? (b) : (a);

return c;
```

Value categories after C++11

TEAMT5 杜浦數位安全

```
int a = 0, b = 0;

int c = (a) ? (b) : (a);

return c;
```

Value categories after C++11



Aiming problem

TEAMT5

Heavy copy cost

https://godbolt.org/z/85E7cM5qz

C++98 -> 482 ms

C++20 -> 214 ms

```
13  std::vector<int> doubleValues (const std::vector<int>& v)
14  {
15      std::vector<int> new_values;
16      new_values.reserve(v.size());
17      for (std::vector<int>::const_iterator iter = v.begin(), end_itr =
18      {
19          new_values.push_back( 2 * *iter );
20      }
21      return new_values;
22  }
23
24  void stuff() {
25      std::vector<int> v;
26      for (size_t i = 0; i < 1000; i++ )
27          v.push_back( i );
28
29      std::vector<int> a = doubleValues( v );
30  }
31
32  int main()
33  {
34      unsigned long long t = timeInMilliseconds();
35      for (size_t i = 0; i < LARGE_NUMBER; i++ )
36          stuff();
37      std::cout << timeInMilliseconds() - t << std::endl;
38      return 0;
39  }
```

```
int a;      // a 為左值
a = 3;      // 3 為右值
```

```
int a = 0;
int b = 42;
int c = a + b;// a b 轉值為右值
```

```
Data a = 0;
Data b = 42;
Data c = a + b;
```

```
Data a = 0;
Data b = 42;
Data c = a + b;
```

1. 定義物件 a，呼叫建構子 Data(int);
2. Data(int val = 0) -> 初始化為 0;

```
Data a = 0;
Data b = 42;
Data c = a + b;
```

1. 取物件 a 的運算子 operator+()
2. b 的型態為 Data (有無**候選**函式)
3. 呼叫 Data::operator+(const Data&);
4. 從 this->val 取值 b.val 取值後相加
5. **建構並初始化**新值 c

```
Data a = 0;
Data b = 42;
Data c = a + b;
```

1. 取物件 a 的運算子 operator+()
2. b 的型態為 Data (有無候選函式)
3. 呼叫 Data::operator+(const Data&);
4. 從 this->val 取值 b.val 取值後相加
5. 建構並初始化新值 c

右值

C++98

```
Data a = 0;
Data b = 42;
Data c = a + b;
```

1. 取物件 a 的運算子 operator+()
2. b 的型態為 Data (有無候選函式)
3. 呼叫 Data::operator+(const Data&);
4. 從 this->val 取值 b.val 取值後相加
5. 建構並初始化新值 c
6. 離開 operator+ **複製回傳值**，並建構 c

Move semantics

What’s the difference?

```
Data a = 0;
Data b = 42;
Data c = a + b;
```

1. 取物件 a 的運算子 operator+()
2. b 的型態為 Data (有無**候選**函式)
3. 呼叫 Data::operator+(const Data&);
4. 從 this->val 取值 b.val 取值後相加
5. **建構並初始化新值 c**

```
Data a = 0;
Data b = 42;
Data c = a + b;
```

1. 取物件 a 的運算子 operator+()
2. b 的型態為 Data (有無**候選**函式)
3. 呼叫 Data::operator+(const Data&);
4. 從 this->val 取值 b.val 取值後相加
5. **建構並初始化新值 c**

移動 operator+ 的結果到函式外部

Move semantics


```
struct Data {
    Data() = default;
    Data(const Data &other) = default;
    Data &operator=(const Data &other) = default;
    Data(Data &&other) = default;
    Data &operator=(Data &&other) = default;
    ~Data() = default;

    int value;
};
```

Default ctor/dtor

TEAMT5
杜浦數位安全

```
struct Data {
    Data() = default;
    Data(const Data &other) = default;
    Data &operator=(const Data &other) = default;
    Data(Data &&other) = default;
    Data &operator=(Data &&other) = default;
    ~Data() = default;

    int value;
};
```

Default constructor
Default destructor

Default ctor/dtor


```
struct Data {
    Data() = default;
    Data(const D
    Data &operat
    Data(Data &&other) = default;
    Data &operator=(Data &&other) = default;
    ~Data() = default;

    int value;
};
```

Since C++11

Default constructor
Default destructor

C++98 default and copy


```
struct Data {
   Data() = default;
   Data(const Data &other) = default;
   Data &operator=(const Data &other) = default;
   Data(Data &&other) = default;
   Data &operator=(Data &&other) = default;
   ~Data() = default;

   int value;
};
```

C++98 default and copy

TEAMT5 杜浦數位安全

```
struct Data {
    Data() = default;
    Data(const Data &other) = default;
    Data &operator=(const Data &other) = default;
    Data(Data &&other) = default;
    Data &operator=(Data &&other) = default;
    ~Data() = default;

    int value;
};
```

左值參考
lvalue reference

C++98 default and copy

TEAMT5 杜浦數位安全

```
struct Data {
    Data() = default;
    Data(const Data &other) = default;
    Data &operator=(const Data &other) = default;
    Data(Data &&other) = default;
    Data &operator=(Data &&other) = default;
    ~Data() = default;

    int value;
};
```

有 & 就是參考 reference
表示要存取該段記憶體位址

左值參考
lvalue reference

Move semantics


```
struct Data {

    Data() = default;

    Data(const Data &other) = default;
    Data &operator=(const Data &other) = default;

    Data(Data &&other) = default;

    Data &operator=(Data &&other) = default;

    ~Data() = default;



    int value;
};
```

Move semantics


```
struct Data {

    Data() = default;

    Data(const Data &other) = default;
    Data &operator=(const Data &other) = default;

    Data(Data &&other) = default;

    Data &operator=(Data &&other) = default;

    ~Data() = default;


    int value;
};
```

兩個, 右值參考

Difference


```
struct Data {

    Data() = defaul
    Data(const Data &other) = default;

    Data &operator=(const Data &other) = default;

    Data(Data &&other) = default;

    Data &opera

    ~Data() = default;

    int value;

};
```

左值參考 不能繫結純右值

右值參考 只繫結右值

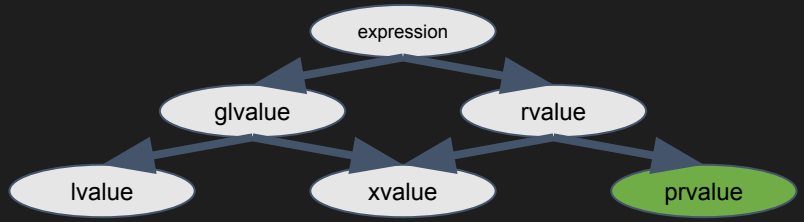
Difference

TEAMT5 杜浦數位安全

```
struct Data {
    Data() = defaul
    Data(const Data &other) = default;
    Data &operator=(const Data &other) = default;
    Data(Data &&other) = default;
    Data &opera
    ~Data() = default;
    int value;
};
```

左值參考 不能繫結純右值

右值參考 只繫結右值



Move semantics

TEAMT5 杜浦數位安全

```
struct Data {

    Data() = default;

    Data(const Data &other) = default;
    Data &operator=(const Data &other) = default;

    Data(Data &&other) = default;

    Data &operator=(Data &&other) = default;

    ~Data() = default;


    int value;
};
```

1. Move 之後的使用是未定義行為
2. Named rvalue reference is an lvalue

Movable Copyable

TEAMT5 杜浦數位安全

C

```
Data(const Data &other) = default;
Data &operator=(const Data &other) = default;
Data(Data &&other) = delete;
Data &operator=(Data &&other) = delete;
```

```
Data(const Data &other) = default;
Data &operator=(const Data &other) = default;
Data(Data &&other) = default;
Data &operator=(Data &&other) = default;
```

M

```
Data(const Data &other) = delete;
Data &operator=(const Data &other) = delete;
Data(Data &&other) = delete;
Data &operator=(Data &&other) = delete;
```

```
Data(const Data &other) = delete;
Data &operator=(const Data &other) = delete;
Data(Data &&other) = default;
Data &operator=(Data &&other) = default;
```

Move semantics


```
struct Data {
    std::string s;
    Data() = default;
    Data(std::string &&s) : s(std::move(s)) {}
    Data &append(Data &&s) {
        this->s += s.s;
        s.s.clear();
        return *this;
    }
    friend std::ostream &operator<<(std::ostream &os, const Data &s);
};
```

https://godbolt.org/z/djqMcPrd3

Move semantics

TEAMT5 杜浦數位安全

```
struct Data {
    std::string s;
    Data() = default;
    Data(std::string &&s) : s(std::move(s)) {}
    Data &append(Data &&s) {
        this->s += s.s;
        s.s.clear();
        return *this;
    }
    friend std::ostream &operator<<(std::ostream &os, const Data &s);
};
```

https://godbolt.org/z/djqMcPrd3

Move semantics


```cpp
int main()
{
    std::string s = "Hello";
    Data s1, s2{std::move(s)};
    s1.append(std::move(s2));

    std::cout << s1 << ":" << s2 << std::endl;

    return 0;
}
```

https://godbolt.org/z/djqMcPrd3

Move semantics

TEAMT5 杜浦數位安全

```cpp
int main()
{
    std::string s = "Hello";
    Data s1, s2{std::move(s)};
    s1.append(std::move(s2));

    std::cout << s1 << ":" << s2 << std::endl;

    return 0;
}
```

移動語意,
在此之後使用 s 都是未定義

https://godbolt.org/z/djqMcPrd3

Move semantics


```cpp
int main()
{
    std::string s = "Hello";
    Data s1, s2{std::move(s)};
    s1.append(std::move(s2));

    std::cout << s1 << ":" << s2 << std::endl;

    return 0;
}
```

移動語意,
在此之後使用 s2 都是未定義

https://godbolt.org/z/djqMcPrd3

Move semantics


```
struct Data {
      std::string s;
      Data() = default;
      Data(std::string &&s) : s(std::move(s)) {}
      Data &append(Data &&s) {
          this->s += s.s;
          s.s.clear();
          return *this;
      }
      friend std::ostream &operator<<(std::ostream &os, const Data &s);
};
```

https://godbolt.org/z/djqMcPrd3

Move semantics

TEAMT5
杜浦數位安全

```
int main()
{
    std::string s = "Hello";
    Data s1, s2{std::move(s)};
    s1.append(std::move(s2));

    std::cout << s1 << ":" << s2 << std::endl;

    return 0;
}
```

未定義行為

https://godbolt.org/z/djqMcPrd3

std::move()


```
template <class _Tp>
_LIBCPP_NODISCARD_EXT inline _LIBCPP_INLINE_VISIBILITY _LIBCPP_CONSTEXPR typename remove_reference<_Tp>::type&&
move(_Tp&& __t) _NOEXCEPT {
 typedef _LIBCPP_NODEBUG typename remove_reference<_Tp>::type _Up;
 return static_cast<_Up&&>(__t);
}
```

std::move()

TEAMT5 杜浦數位安全

```
template <class _Tp>
_LIBCPP_NODISCARD_EXT inline _LIBCPP_INLINE_VISIBILITY _LIBCPP_CONSTEXPR typename remove_reference<_Tp>::type&&
move(_Tp&& __t) _NOEXCEPT {
 typedef _LIBCPP_NODEBUG typename remove_reference<_Tp>::type _Up;
 return static_cast<_Up&&>(__t);
}
```

其實 std::move 只有強制轉型

std::move()


```
template <class _Tp>
_LIBCPP_NODISCARD_EXT inline _LIBCPP_INLINE_VISIBILITY _LIBCPP_CONSTEXPR typename remove_reference<_Tp>::type&&
move(_Tp&& __t) _NOEXCEPT {
 typedef _LIBCPP_NODEBUG typename remove_reference<_Tp>::type _Up;
 return static_cast<_Up&&>(__t);
}
```

但是語意上已經轉移擁有權

std::move()


```
template <class _Tp>
_LIBCPP_NODISCARD_EXT inline _LIBCPP_INLINE_VISIBILITY _LIBCPP_CONSTEXPR typename remove_reference<_Tp>::type&&
move(_Tp&& __t) _NOEXCEPT {
 typedef _LIBCPP_NODEBUG typename remove_reference<_Tp>::type _Up;
 return static_cast<_Up&&>(__t);
}
```

The named rvalue reference is an lvalue
該函式取得擁有權之後，可以任何操作

Move semantics

TEAMT5 杜浦數位安全

```
int main()
{
    std::string s = "Hello";
    Data s1, s2{std::move(s)};
    s1.append(std::move(s2));

    std::cout << s1 << ":" << s2 << std::endl;

    return 0;
}
```

語法合法，**語意**不合法

https://godbolt.org/z/djqMcPrd3

Move semantics

TEAMT5 杜浦數位安全

```
int main()
{
    std::string s = "Hello";
    Data s1, s2{std::move(s)};
    s1.append(std::move(s2));

    std::cout << s1 << ":" << s2 << std::endl;

    return 0;
}
```

語法合法
因為 std::move 其實只有轉型

https://godbolt.org/z/djqMcPrd3

Take away

- Xvalue is an lvalue and also a rvalue.
- prvalue, lvalue, glvalue, rvalue
- std::move change your **ownership**.
- rvalue reference only associate with prvalue.
- Use **const T&** if you can.

Thanks

scc@teamt5.org

TEAMT5

杜 浦 數 位 安 全

Persistent Cyber Threat Hunters