System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++

3-1. Rvalue & Move Semantics

Eugene Zouev

Rvalue references

Move semantics

Move constructors

Move assignment operators

Classic Assignment

Name = Expression

Something that has a precise memory location
lvalue

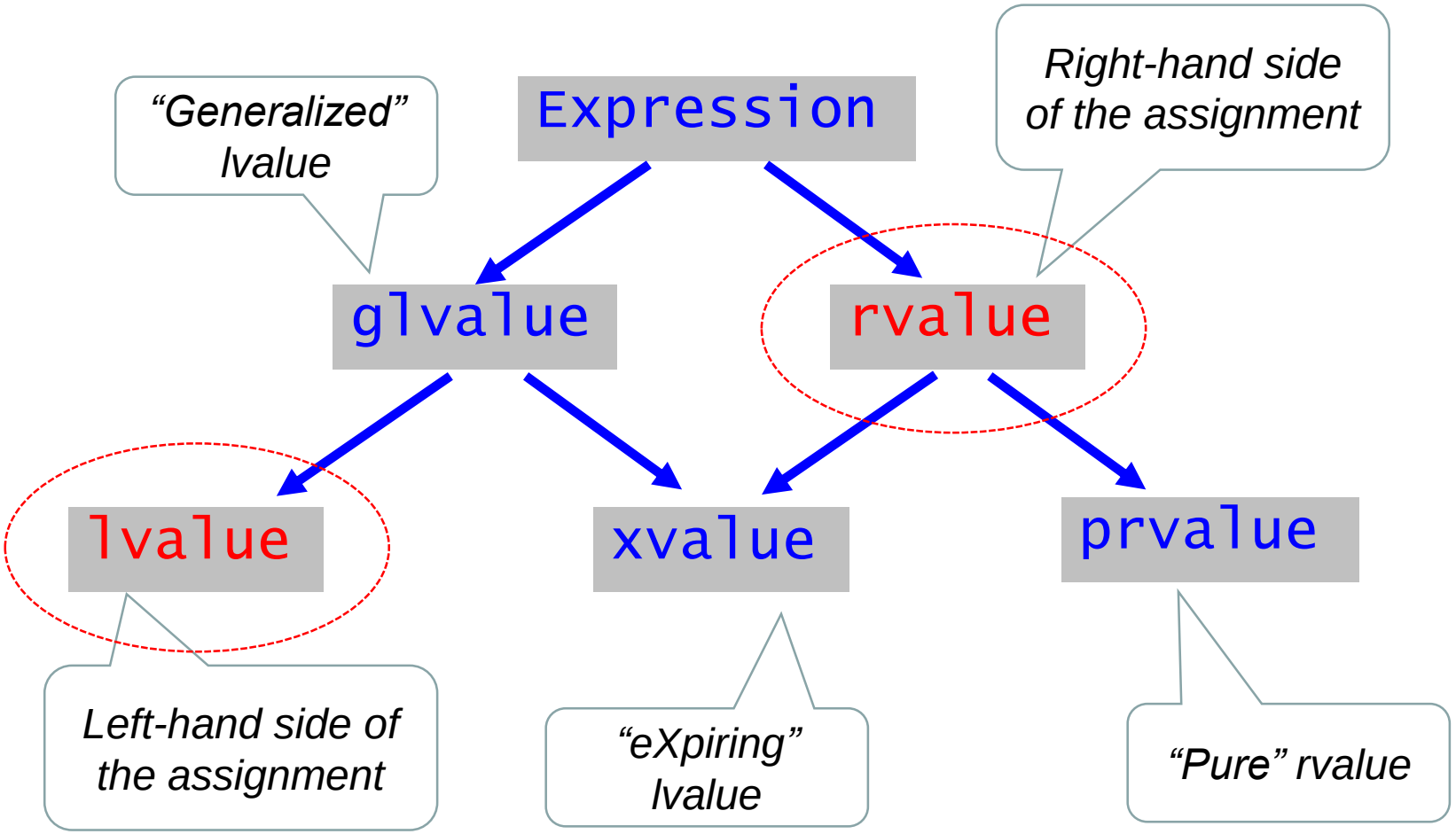
A temporary object: something that is a temporary value
rvalue

```
a    // a simple variable
x[i] // an array element
r    // a reference
f()  // function returning reference
...
```

```
a+1         // an expression
1+2*(5+3) // a constant expression
g()  // function returning a value
...
```

Expression Category Taxonomy

ISO Standard, Sect. 6.10



Expression Category Taxonomy

ISO Standard, Sect. 6.10

- **lvalue**: an expression whose evaluation determines the identity of an object, bit-field, or function.
- **rvalue**: an expression whose evaluation initializes an object or a bit-field, or computes the value of an operand of an operator…
- lvalue, rvalue classify *expressions*, not *values*

Classic Assignment

```
int a;
...
a = x * ( y + z );
```

`x*(y+z)` is a **full expression**; this is a *temporary value* used for assigning to an lvalue.

Here, `a` is **lvalue**: it is a declared entity, and the compiler allocated memory for `a`'s value

`y+z` is a **subexpression**; this is a *temporary value* used for further calculations

Lvalues & Rvalues: an Example

Lvalues are not necessarily variables

```
int x;
int& getRef() { return x; }
...
getRef() = 4;
```

Here, `getRef()` is an **lvalue**

BTW: what does this return?

```
&getRef()
```

Here, `getVal()` is an **rvalue**

```
int x;
int getVal() { return x; }
...
getVal() = 4; // Error
```

Three kinds of initialization

```
T x = expr;
```

Variable initialization

```
void foo(T x);
foo(expr);
```

Argument passing

```
T foo()
{
  return expr;
}
```

Returning value

All the initializations are semantically equivalent

Rvalue references (1)

Since C++11

The key notion for today ☺

```
A a;
A& ref = a;
```

a is lvalue;
ref is the reference to lvalue
- **lvalue reference**

```
A a;
A& ref = A();
```

Error!!!
A() is a temporary object (rvalue)
it's illegal to bind rvalue to a reference to **non-const**. WHY?...

```
A a;
const A& ref = A();
```

OK
It's legal to bind rvalue to a reference to **constant**. WHY?...

Rvalue references (2)

```
A a;
A& ref = A();
```

Error!!!
`A()` is a temporary object (rvalue)
it's illegal to bind rvalue to a reference to **non-const**. WHY?...

The counterexample ☺ from VS:

```
vector<int>& v = vector<int>{ 1, 2, 3 };
v.push_back(4);
```

warning C4239: nonstandard extension used
note: A non-const reference may only be bound to an lvalue

Rvalue references (3)

```
A a;
const A& ref = A();
```

This is (was) the only possible way to pass
a temporary object – e.g., to copy constructor:

```
A::A(const A&);
A a = A();
```

Rvalue references (4)

```
A a;
A& ref = A();
```

Error!!!

```
A&& ref = A();
```

OK!!!

Rvalue reference

Example

```
string foo();
string&& str = foo();

int&& i = 5;
```

Rvalue reference is the means to refer to a temporary object – i.e., to bind a reference with a temporary object

Rvalue and lvalue references

A complete scheme

```
T& ref = lvalue;
```

The usual (lvalue) reference

```
T& ref = rvalue;
```

Illegal: cannot bind rvalue with a reference to a non-const

```
T&& ref = lvalue;
```

Illegal: cannot bind lvalue with an rvalue reference

```
T&& ref = rvalue;
```

The usual rvalue reference

The conclusion is that the following is **OK for all cases**:

```
const T& ref = expression;
```

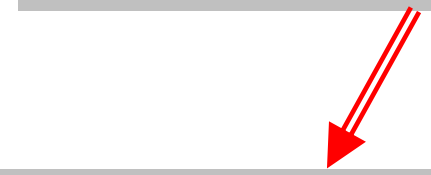
So, the question: Why &&?? What it is all for???

Rvalue references (5)

Minor technical extension to C++11

The idea behind lvalue & rvalue references is to explicitly separate different cases:

```
const T& ref = expression;
```



```
T& ref = lvalue;
```

```
T&& ref = rvalue;
```

Remember separation between `const` & `constexpr`?

The most important aim of such a separation is to support **move semantics**

Reminder: function overloading

```
T
T&
const T&
T&&
const T&&
```

These are different types…

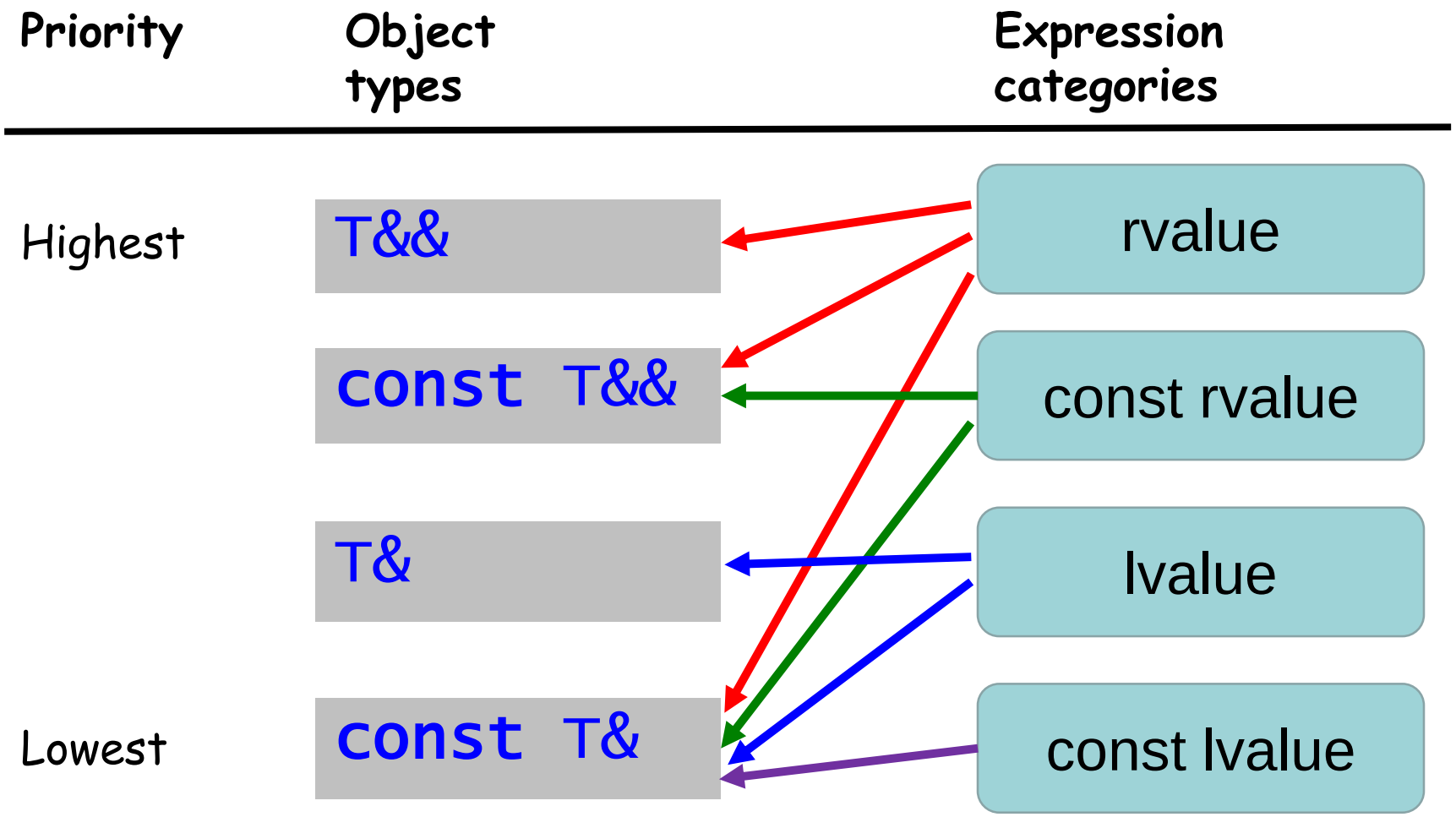
…therefore, they participate in function overloading

```
void foo(T);
void foo(T&);
void foo(const T&);
void foo(T&&);
void foo(const T&&);
```

These are different functions

The task:
Write a program that invokes all these kinds of `foo`.

Reference types: preferences



And the same is obviously about ***constructors***

Move semantics (1)

```
T();                // default constructor
T(const T&);        // usual copy constructor
T(T&);              // dangerous copy ctor
T(T&&);             // move constructor
T(const T&&);       // strange move constructor
```

The same is about assignment operators

```
T(const T&);
```

The usual copy constructor.
Its purpose is to create an object by copying an existing object (without modifying it).
Has the lowest priority, but accepts all reference types.

```
T(T&);
```

The "dangerous" copy constructor.
Creates an object by copying an existing object; can modify it.
Rarely used.

Move semantics (2)

```
T(T&&);
```

The **move constructor**.
Is invoked for **non-const rvalues** – i.e., for temporary objects.
Its purpose is to create an object by moving the contents of the temporary object (which is to be destroyed soon).
Moving assumes that the contents of the argument should be modified (nulled).
Has the highest priority.

```
T(const T&&);
```

The "strange" move constructor.
Its purpose is to move the contents of its argument with nullifying it but it's not possible because of argument's const'ness.
Doesn't make any sense.

Move semantics (3)

Conceptual example

```
template<typename T>
class Array
{
  private:
    T* ptr;
    int size;
  public:
    Array(int sz) : size(sz), ptr(new T[size]) { }
    virtual ~Array() { delete[] ptr; }
    // Copy constructor
    Array(const Array& arr) : Array(arr.size) {
        for ( int i=0; i<arr.size; i++ ) ptr[i] = arr.ptr[i];
    }
    // Move constructor
    Array(Array&& arr) {
        ptr = arr.ptr; arr.ptr = nullptr;
        size = arr.size; arr.size = 0;
    }
}
```

The task:

- Fix all bugs ☺.
- Add copy & move assignment operators
- Test the code for all possible cases.

Move semantics: references

- C++ ISO Standard, Sect. 15.8.1.
- http://en.cppreference.com/w/cpp/language/move_constructor
- https://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html
- https://blog.smartbear.com/c-plus-plus/c11-tutorial-introducing-the-move-constructor-and-the-move-assignment-operator/
- https://habrahabr.ru/post/322132/ (Russian)
- https://habrahabr.ru/post/226229/ (Russian)