Value Categories in C++

Xing GUO | higuoxing@gmail.com

In C++11, with the introducing of *move semantics*, C++ extended its value categories. Hence, it's possibly not a good idea to roughly identify expression's value categories to be lvalue and rvalue. This post is to dump some notes about "Value Category" of expressions in C++, together with some interesting history. Just some basic knowledge about C++ is required.

Expressions

An expression is a sequence of operators and their operands, that specifies a computation. [1]
A program is just a combination of expressions, for example

```
int a = 1;
int b = 2;
a = a + b;
printf("%d", a);
```

We assign *literal 1* to *a*, then assign *literal 2* to *b*, then assign *a+b* to *a*, and then print the result of *a*. (Note: all steps are expressions. function *printf* is an expression as well, which returns the length of the string. In this case, it will return 1, the length of "3", but we did not assign it to anything).

History

Usually, we characterized an expression in two dimensions, *type* and *value category*. In early days, the taxonomy of value categories was first introduced in CPL (Combined Programming Language, the ancestor of BCPL, Basic Combined Programming Language). In CPL, expressions can be evaluated in two modes, known as *left-hand* (LH) and *right-hand* (RH) modes. However, only certain kinds of expression are meaningful in LH mode. Take this factorial computing function for example:

```
function Fact2[x] = result of
  § real f  = 1
    until x = 0 do
      f, x := xf, x -- 1
    result := f §
```

The CPL looks like some kinds of pseudocode, so here, we will not give too much details about its grammars. In the expression *f, x := xf, x -- 1* (*xf* is similar to what we do *x*f* in C/C++), the *assignment command* := is to tell the computing system to compute the *RH value* obtained from right-hand of the symbol and then assign the *RH value* to the *address* (*LH value*) that obtained from left-hand of the assignment symbol (:=). Hence, this is the original version taxonomy of value categories and that is what generally in our minds about lvalue and rvalue. For further features about CPL, I recommend "The main features of CPL" [2]
.

[1] From cppreference
[2] The main features of CPL

Then, in C, Dennis Ritchie (or dmr, father of C programming language and co-author of Unix system) characterized expressions into *lvalues* and *others*. lvalue is a “locator value”. This is very similar to the taxonomy in CPL.
In early C++, Bjarne Stroustrup did the same as C, but restored the term “rvalue” for “non-lvalue expressions”. Besides, in C++98, functions are made into lvalues, references could be bound to lvalues, and only reference to const value could be bound to rvalue (we do not need to think too much about *reference to const* that bound to rvalue, we will come to it later). A small example:

```
int global_val = 0;

int& func() {
  return global_val;
}

int main() {
  func() = 2;
  return 0;
}
```

In this example, function *func* return the reference of *global_val* which is a lvalue (function that returns a lvalue reference is a lvalue), and we could assign *literal 2* to the reference returned by *func*.
In C++11, just like what we mentioned before, expressions have two properties: *has identity* and *“can be moved from”*. If an expression *“has identity”*, we could tell if two copies are identical. For example, *int a = 2; int b = 2;* we know that *a* and *b* are identical, because they are two variables and have different addresses in memory, while values of literals like (char) *‘a’*, (int) *12 “do not have identity”*. The property *“can be moved from”* denotes whether the resource managed by this object can be transferred to another object. This is related to move semantics in C++11, which we will discuss in next section. Based on these two independent properties, expressions could be categorized into four groups[3]:

- im -- *“has identity” and “can be moved from”*, named to be *xvalue*
- iM -- *“has identity” and “cannot be moved from”*, named to be *lvalue*
- Im -- *“do not have identity” and “can be moved from”*, named to be *prvalue*
- IM -- *“do not have identity” and “cannot be moved from”*

However, the last one is not existed in C++ or most of programming languages. Hence, the relationships between im, iM, Im can be described in one diagram:

```
(iM)    (im)    (Im)       (lvalue)  (xvalue)  (prvalue)
   \   /    \   /      or      \        /     \      /
    (i)      (m)              (glvalue)     (rvalue)
```

Note: glvalue means *generalized lvalue*, *xvalue* means *eXpiring value* and *prvalue* means *pure rvalue*.

[3] The i and m notations are from Bjarne Stroustrup post [*“New” Value Terminology*]

Here, we just give the names of value categories in C++11, for their details, we have to give them after explaining the move semantics in next section.

Move Semantics & RValue Reference
Move Semantics was introduced in C++11, and was designed to lower the cost when transferring the ownership of resources. For example, we have a big object who owns huge external resources. So, if we want to transferring the ownership of resources, in C++, we could simply make the new owner's resource pointer point to the same memory area, rather than make a copy of the resource and then delete the original one.

```cpp
#include <iostream>

class BigObject {
private:
  int *_resource;
  size_t _size;
public:
  BigObject(size_t size)
    : _size(size), _resource(new int[size]) {
    printf("create an object with %zu resources\n", _size);
  }

  BigObject(const BigObject& another)
    : _size(another._size), _resource(new int[another._size]) {
    for (int i = 0; i < another._size; ++ i) {
      _resource[i] = another._resource[i];
    }
    printf("copy an object with %zu resources\n", _size);
  }
  ~ BigObject() {
    delete [] _resource;
    printf("destory an object with %zu resources\n", _size);
  }
};

BigObject createBigObject(size_t size) {
  BigObject tmp(size);
  return tmp;
}

int main() {
  BigObject bigObj(createBigObject(100));
  return 0;
}
```

In this example, we have created a *BigObject* and it has a dynamic *int array*. Note: most of modern compilers will inspect our return values, and will do some kinds of optimizations called *RVO -- (Return Value Optimization)* or *NRVO -- (Named Return Value Optimization),* this could help prevent constructing useless temporary variables. In this case, *BigObject tmp* is a temporary variable returned by function *createBigObject()*, and *tmp* was created by *BigObject()*. So, if we compile the code simply with

```
clang++ -std=c++11 main.cc && ./a.out
```

compiler will just pass the original data to *bigObj* and then we could get the printed messages:

```
create an object with 100 resources
destory an object with 100 resources
```

Adding the flag “-fno-elide-constructors” could tell the compiler not to perform the optimizations. Here, we will not dig too much about *RVO* or *NRVO*.

```
clang++ -std=c++11 -fno-elide-constructors main.cc && ./a.out
```

then we will get

```
create an object with 100 resources
copy an object with 100 resources
destory an object with 100 resources
copy an object with 100 resources
destory an object with 100 resources
destory an object with 100 resources
```

As you see, our data was copied two times and it is not difficult to understand. *BigObject tmp(100);* created the original data, then original data was copied to *tmp*, and then *tmp* was copied to *bigObj*. Because function *createBigObject* returned rvalue and when the code run out of the scope, it will disappear, and what we could do is to copy it again. How to solve this problem? One simple solution is to return the instance of object by pointer. However, in C++0x, it provides us a new feature called *rvalue reference*. It means we could pass a rvalue reference out of its scope. Note: *BigObject&&* means *rvalue reference* not *reference to reference*.

```
BigObject(BigObject&& another) // move constructor
  : _size(another._size), _resource(new int[another._size]) {
  _resource = another._resource;
  another._resource = nullptr;
  another._size = 0;
  printf("move an object with %zu resources\n", _size);
}
```

With this, our compiler will automatically using this *move constructor* to pass the ownership without copying the resources again and again. This will print:

```
create an object with 100 resources
move an object with 100 resources
destory an object with 0 resources
move an object with 100 resources
destory an object with 0 resources
destory an object with 100 resources
```

This is a simple example of how *moving semantics* help improve the performance of codes. Actually, the property *"can be moved from"* is derived from this. C++ STL provides us a useful function called *std::move()*, which could help us transfer ownership efficiently.

```
string a0 = "hello, world";
string a1 = std::move(a0);
```

Now, we could discuss the taxonomy.

- **lvalue** -- *"have identity"*, *"cannot be moved from"*. Because if we want to modify or use a lvalue later, the ownership of it cannot be stolen or moved from, this will cause catastrophe.
  e.g.

  ```
  int a = 1;
  /* a is a lvalue, because we may reference it later */
  ++ a; -- a;
  /* ++ a or -- a are lvalues, because a will immediately
  increase/decrease by 1, and we may use the value of a in the
  context later. */
  "hello";
  /* Yes, literal string is a lvalue, because it cannot be moved
  from, and it is array type, two strings "hello", "hello" may
  refer to two different memory area */
  ```

- **prvalue** -- *"do not have identity", "can be moved from". an expression whose evaluation either computes the value of the operand of an operator, or initializes an object or a bit-field.*
  e.g.

  ```
  a + b;
  /* expression that computing the value of a + b and left no
  object */
  43;
  /* int literal */
  true;
  /* bool literal */
  ```

- **xvalue** -- *"has identity", "can be moved from". a glvalue that denotes an object or bit-field whose resources can be reused*
  e.g.

  ```
  std::move(x);
  /* will be expiried later */
  a.m
  /* member of object expression */
  ```

- **glvalue** -- *"has identity". an expression whose evaluation determines the identity of an object, bit-field, or function.*
- **rvalue** -- *"can be moved from". Could appear on the righ-hand side of an assignment expression. So-called, historically*

Conclusion

All these value categories are designed to help us construct high performance and robust codes, help compiler whether, when and where to perform optimizations. Last thing to remember, moving action is not happening when *std::move* is called but *move constructor* is called.

Further Reading

[1] A Taxonomy of Expression ValueCategories
[2] "New" Value Terminology
[3] RVO V.S. std::move
[4] Cpp reference - Value Category