Value Categories in C++

Xing GUO | higuoxing at gmail dot 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 category to be lvalue or rvalue for historical reasons. This post is to dump some notes about "Value Category" of an expression in C++, combined with some interesting history. Just some basic knowledges about C++ are 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*, then print the result of *a*. (Note: all the steps are expressions. function *printf* is expression as well and will return the length of string you want print. In this case, it will return 1, the length of "3", but we did not assign it to anything).

Value Categories

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 will not give too much words on its grammar. 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 assign the *RH value* to the *address* (*LH value*) that obtained from left-hand of the symbol. Hence, this's the original version taxonomy of value categories and that's 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 expression into *lvalue* and *others*. lvalue is a “locator value”. This is very similar to the taxonomy in CPL.
Then, 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 don’t 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, and we could assign *literal 2* to the reference.
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 address in memory. However values of literals like (char) *‘a’*, (int) *12* don’t have identity, because they have no difference. 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 name 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 it 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.

```
#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, when run this code. 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 out to *bigObj* and then we could get the printed messages:

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

Adding 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’s 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 address a rvalue using rvalue reference. *BigObject&&* means *rvalue reference* not *reference to reference*.

```
BigObject(BigObject&& another)
  : _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. So, now we could talk about the taxonomy.

Taxonomy

Previous examples give us an overview on *move semantics*, however, it's not always a good thing (e.g. we want make a copy of the original data to another object, but the code moving the data to the another object. When we want to modify the original object, the original object has been a blank one). This is an example of expression *has identity* but *cannot moved from* (iM), under such circumstance, we'd better not perform moving ownership of resources. We categorize these expressions to be *lvalues* (e.g. pointers). Expressions *do not have identity* and *can be moved from* is called *prvalue* (e.g. the calling of function whose return type is not a reference, literals 11.2, 'a', true and so on). Expressions *do not have identity* and *can be moved from* is *xvalue* (e.g. the calling of function whose return type is rvalue reference, because it will be eXpired later). Then, expressions those who *has identity* is *glvalue*, and those who *can be moved from* is *rvalue*.

Further Reading

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