Subject: How would you design a good copy/move semantics system?
Posted by cbpporter on Mon, 15 Dec 2014 14:08:38 GMT
View Forum Message <> Reply to Message

This is more of a question for people deeply familiar to the way copy-constructing works.

I was thinking of a system where:
- each class can have a copy and a move optionally
- copy works like the default copy constructor, except for classes where deep copies are needed
- move works pretty much the way it works in U++, but only destroys data that would be copied by a deep copy
- calling move on a class that does not have a move implemented will do a copy
- the rules apply based on class depth

This are the principles. I need to also do an implementation that has as low overhead as possible performance wise and that does not look particularly ugly.

Subject: Re: How would you design a good copy/move semantics system?
Posted by cbpporter on Thu, 18 Dec 2014 11:55:42 GMT
View Forum Message <> Reply to Message

What I'm mostly interested in is what are the rules for guaranteeing a maximum invocation of move constructors in implicit situations, like when assigning or returning values.

U++ chose to have default copy to be move for this reason, right?

Subject: Re: How would you design a good copy/move semantics system?
Posted by Lance on Fri, 19 Dec 2014 23:13:20 GMT
View Forum Message <> Reply to Message

Hi cbpporter:

I am not sure if I understand you correctly, but it seems to be the way it is in C++11.

--each class can have a copy and a move optionally

true.

--copy works like the default copy constructor, except for classes where deep copies are needed
most time you can use =default to provide a default copy constructor. If not, you supply a copy ctor body.

- move works pretty much the way it works in U++, but only destroys data that would be copied by a deep copy
move behavior is defined by the function body you supply. You may tailor it to behave exactly like

what Upp does, you can do it otherwise.

--calling move on a class that does not have a move implemented will do a copy
that's the way it is.

```
#include <iostream>

using namespace std;

class C
{
public:
 C(){}
 C(const C& c){ cout<<"copy ctor of C invoked."<<endl;}
};

class CM
{
public:
 CM(){}
 CM(const CM& c){ cout<<"copy ctor of CM invoked."<<endl;}
 CM(CM&& c){ cout<<"move ctor of CM invoked."<<endl;}
};

C&& ReturnC(){ return C(); }
CM&& ReturnCM(){ return CM(); }

int main()
{
 C c=ReturnC();
 CM cm=ReturnCM();
}
```

--the rules apply based on class depth
basically true. copy ctor will always call base class copy ctor. Not so with move ctor. The case with move ctor is a little bit interesting.

```
#include <iostream>

using namespace std;

class CM
{
public:
 CM(){cout<<"CM default ctor invoked"<<endl;}
```

```
 CM(const CM& c){ cout<<"copy ctor of CM invoked."<<endl;}
 CM(CM&& c){ cout<<"move ctor of CM invoked."<<endl;}
};

class CMD : public CM
{
public:
 CMD(){cout<<"CMD default ctor invoked"<<endl;}
 CMD(const CMD& c){ cout<<"copy ctor of CMD invoked."<<endl;}
 CMD(CMD&& c){ cout<<"move ctor of CMD invoked."<<endl;}

};


int main()
{
 CMD cm;

 cout<<"------------------------"<<endl;

 CMD cmd=std::move(cm);
}
```

Here is the result of running the above program:

```
CM default ctor invoked
CMD default ctor invoked
------------------------
CM default ctor invoked
move ctor of CMD invoked.
```

This is not right. The right way to do it is like this:

```
#include <iostream>

using namespace std;

class CM
{
public:
 CM(){cout<<"CM default ctor invoked"<<endl;}
 CM(const CM& c){ cout<<"copy ctor of CM invoked."<<endl;}
 CM(CM&& c){ cout<<"move ctor of CM invoked."<<endl;}
};

class CMD : public CM
{
public:
```

```
 CMD(){cout<<"CMD default ctor invoked"<<endl;}
 CMD(const CMD& c){ cout<<"copy ctor of CMD invoked."<<endl;}
 CMD(CMD&& c):CM(std::move(c)){ cout<<"move ctor of CMD invoked."<<endl;}
};

int main()
{
 CMD cm;

 cout<<"------------------------"<<endl;

 CMD cmd=std::move(cm);
}
```

And here is the result of running it

```
CM default ctor invoked
CMD default ctor invoked
------------------------
move ctor of CM invoked.
move ctor of CMD invoked.
```

And if you are concerned with Upp containers, this post may be of interests to you:

http://www.ultimatepp.org/forums/index.php?t=msg&th=8448 &start=0&

Subject: Re: How would you design a good copy/move semantics system?
Posted by cbpporter on Wed, 07 Jan 2015 10:31:30 GMT
View Forum Message <> Reply to Message

Hi Lance,

Back from my holidays so Happy New Year to you and everyone else here!

Quote:
CMD(CMD&& c):CM(std::move(c))

I have very little practical experience with C++11 so I did not know that you need to explicitly call the move constructor as such when using inheritance.

What I'm trying to obtain is basically a single rule/library wide convention relating to move, without C++11 features. Things become more complicated when assignment operators are added and

also when the code gets optimized, with some assignments not having code generated for them.

Not sure yet if one convention is enough to cover all cases. Furthermore, not sure if there are some implementation constrains related to C++ in order to maximize implicit call of move when returning values.

Subject: Re: How would you design a good copy/move semantics system?
Posted by Lance on Thu, 08 Jan 2015 00:16:58 GMT
View Forum Message <> Reply to Message

Hi cbpporter:

Happy New Year!

c++11 is here and will stay. The features you required are part of c++11, which means you can use it without any(almost) extra effort.

The code you quoted is like this: if base class have a copy constructor, derived class' move constructor will use the base class copy constructor to construct the base part of a derived object unless explicitly delegated to another ctor. Sounds very complicated, maybe it's easier to use an example:

```
struct base
{
   base() : buff(nullptr), buff_len(0u){}

   // copy ctor
   base(const base& b): buff_len(b.buff_len)
   {
      if(buff_len)
      {
          buff=new char[buff_len];
          memcpy(buff,b.buff,buff_len);
      }else
          buff=nullptr;
   }

   // move ctor, essentially do what Upp-pick is supposed to do
   base(base&& b):buff_len(b.buff_len), buff(b.buff)
   {
         b.buff_len=0u;
         b.buff=nullptr;
   }

private:
   char * buff;
```

```
    unsigned buff_len;
};

struct derived : public base
{
    derived(): i(0){} // will call base::default ctor to consturct base part of *this;

    derived(const derived& d) : i(d.i) {} // will call base::copy ctor to construct base part of *this;

    derived(derived&& d): i(d.i){} // you may expect base::move ctor to be called to construct base
part of *this.
                // I do think the c++ committee should default to use base move ctor for derived
move ctor.
                // unfortunately, this is not the case. you have to explicitly delegate the construction
of
                // the base part of *this to base move ctor, with something like this:
                //
                //  derived(derived&& d) : base(std::move(d)), i(d.i){}
                //
                // this is the point I was trying to make.

private:
    int i;
};
```

HTH.

Lance

Subject: Re: How would you design a good copy/move semantics system?
Posted by mirek on Thu, 08 Jan 2015 10:23:21 GMT

View Forum Message <> Reply to Message

cbpporter wrote on Thu, 18 December 2014 12:55What I'm mostly interested in is what are the rules for guaranteeing a maximum invocation of move constructors in implicit situations, like when assigning or returning values.

U++ chose to have default copy to be move for this reason, right?

Originally, yes. But please note that the whole thing changed last year.

Now you have to be explicit, either 'pick' or 'clone' the source, with some exceptions (e.g. returning temporary). (Enforced in C++11, in C++0x it is backward compatible).

IMO, C++11 way, where move constructor is called "sometimes" based on context leads to using

deep copy in unexpected situations.

Subject: Re: How would you design a good copy/move semantics system?
Posted by cbpporter on Mon, 12 Jan 2015 10:21:49 GMT
View Forum Message <> Reply to Message

I was interested in this in general, not necessarily in the context of C++, but in the context of value based programming languages.

Ideologically, I shall try a system that always does a copy, including on parameter passing, except on a return statement and see if that takes me somewhere. And an undocumented move operator that can be used but there aren't any hard rules or situations when you absolutely need it.

Subject: Re: How would you design a good copy/move semantics system?
Posted by mirek on Mon, 12 Jan 2015 18:06:21 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 12 January 2015 11:21I was interested in this in general, not necessarily in the context of C++, but in the context of value based programming languages.

Well, ideologically speaking, I would say normally there are 2 types of entities:

- those than can be (deeply) copied, usually called "concrete types". Typically, not polymorphic, no abstraction. E.g. Color, Font, Value...
- those that do not have easy meaningful copy - say "identity types". Usually all polymorphic types fall into this category. E.g. File, Ctrl, ...

I like to say that U++ adds to the list 3rd type:

- containers. For these it makes sense to have both 'clone' (in case that they contain concrete types) and 'pick' semantics. 'pick', besides being optimization, nicely adds RAII compatible and MT friendly way to move identity types around.