Subject: Should the pick semantics be changed?
Posted by piotr5 on Tue, 04 Mar 2014 10:00:59 GMT
View Forum Message <> Reply to Message

c++11 introduced the r-value notion with "&&" after the type. this means that in namespace std picking is default only for those r-values, deep-copy otherwise. to force picking you have to use the function std::move(). this is quite the opposite to u++ where picking always is default and deep copy must be enforced. what about a redesign to make use of the way std containers handle picking? of course such a rewrite wouldn't work with non-c++11 compilers. as far as I remember at the beginning of u++ old compiler versions were not fully supported either, so it might be a good idea to start coding up some Core lib 2.0 and rewrite all applications...

needless to say, before we do that, first the u++ code parser needs to be made ready for parsing c++11 code, so Assist and documentation will work too...

also interesting for u++ is the suffix to strings, thereby allowing for automatically generated String objects and for applying translations with less bracket-usage. I guess many more improvements could be made if u++ would break backwards-compatibility and focus on the new c++11 features...

Subject: Re: Should the pick semantics be changed?
Posted by mirek on Tue, 04 Mar 2014 11:12:28 GMT
View Forum Message <> Reply to Message

piotr5 wrote on Tue, 04 March 2014 05:00c++11 introduced the r-value notion with "&&" after the type. this means that in namespace std picking is default only for those r-values, deep-copy otherwise. to force picking you have to use the function std::move(). this is quite the opposite to u++ where picking always is default and deep copy must be enforced. what about a redesign to make use of the way std containers handle picking? of course such a rewrite wouldn't work with non-c++11 compilers. as far as I remember at the beginning of u++ old compiler versions were not fully supported either, so it might be a good idea to start coding up some Core lib 2.0 and rewrite all applications...

I am well aware about '&&' and pondering this too. The problem is that it does not have (AFAIK) any composition rules:

```
struct Bar {
  int foo;
  Vector<int> bar;
};
```

Now in U++ Bar has pick semantics without doing anything. To define proper move "&&" constructor/copy, you would need to provide the code to copy all elements one by one. That might be quite tedious error-prone for classes with a large number of elements.

Also, the decision to make pick the implicit variant for containers goes a bit deeper. Consider

Array<Ctrl> ctrl;

now how would you define 'deep copy' of this?

Quote:
needless to say, before we do that, first the u++ code parser needs to be made ready for parsing c++11 code, so Assist and documentation will work too...

also interesting for u++ is the suffix to strings, thereby allowing for automatically generated String objects and for applying translations with less bracket-usage. I guess many more improvements could be made if u++ would break backwards-compatibility and focus on the new c++11 features...

Yes. There are some really nice features in C++11 and I will be happy to use them someday. Parser needs to be fixed for C++11 anyway (actually, I guess there is some need to fix "old" C++ too

That said, I do not think C++11 would fundamentally change the way I write software. For me, basically all needed is already there in C++98...

BTW, I just wonder, what do you mean by "suffix" to strings?

Mirek

Subject: Re: Should the pick semantics be changed?
Posted by piotr5 on Tue, 04 Mar 2014 22:14:55 GMT
View Forum Message <> Reply to Message

have you tried
struct Bar {
int foo;
Vector<int> bar;
Bar(Bar&&)=default;
};?

Array<Ctrl> is imho a bad idea to make public, so you just need to define a pick-constructor and delete the copy-constructor. I think with the syntactic sugar of c++11 the original reasons for the current design of pick semantics are becoming obsolete.

with the suffix stuff I am referring to the operator""_suffix() syntax. to call that function you just write any literal and add _suffix at the end. of course you can replace _suffix by whatever name. (additionally there is the constexpr keyword to create objects at compile-time.) so I imagine instead of writing T_("text") next version of Core could also accept "text"_T to perform translations...

I experimented a bit with c++11 and I noticed the programming-style is quite different: std::array sounds more useful than the old c-style arrays, I keep using std algorithms more frequently because function-objects are easier to write, and most object-initialization I am doing with initializer-lists. but I cannot say for sure how this language will evolve. maybe someone will have a nice idea for new language-features by using the new stuff...

Subject: Re: Should the pick semantics be changed?
Posted by mirek on Wed, 05 Mar 2014 07:54:57 GMT
View Forum Message <> Reply to Message

piotr5 wrote on Tue, 04 March 2014 17:14have you tried
struct Bar {
int foo;
Vector<int> bar;
Bar(Bar&&)=default;
};?

Aah, I have to admit I have missed this one...

10 years ago I was discussing && with its original author (Howard Hinnant) over usenet (http://www.archivum.info/comp.lang.c++.moderated/2005-08/00708/Re-Rvalue-references--a-done-deal.html) and at that time, the sentiment was against providing this. I am glad it got through in the end.

Well, in this case I believe && could really replace pick_ soon. I will be really be only glad for this to happen... It will be safer and more in accord with C++ mainstream.

But I still believe it is a good idea to keep "pick" behaviour as default (regardless it is implemented as && or ugly pick_ macro) and operator<<= for deep copy. Also, I found over years extremely useful to maintain the source in picked state (not to clear it, as is AFAIK common for C++11 usage).

The only issue I can see now is that for pick types, you have to define twice as much constructors and operator= to keep the current behaviour, you need

T(&)
T(&&)
operator=(&)
operator=(&&)

But that is only a small price to pay...

The other bad thing is that I unfortunately still have to support the "old" C++ now...

That said, it would perhaps be possible to have U++ "dual-mode", using move constructors with

C++11 and pick_ with old C++.

Mirek

Subject: Re: Should the pick semantics be changed?
Posted by piotr5 on Wed, 05 Mar 2014 09:27:29 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 05 March 2014 08:54
But I still believe it is a good idea to keep "pick" behaviour as default (regardless it is implemented as && or ugly pick_ macro) and operator<<= for deep copy. Also, I found over years extremely useful to maintain the source in picked state (not to clear it, as is AFAIK common for C++11 usage).
I agree, for now when compiling in c++11 the operator=(&) should produce a compilation error whenever instantiated, informing the programmer that picking is default and right side either must be enclosed in std::move (or maybe rather some customized Upp::Pick) or use operator<<= for deep copy. would break some old code (but only when compiling c++11) but is more user-friendly.
however, for constructor, in c++11 please let us add a (deep-)copy-constructor and initialization list constructor, preferably both enclosed in a single macro, maybe included in the std-compatibility-stuff...

that way backwards-compatibility wouldn't break, instead c++11-users would encounter a different interface to U++, one where deep-copy constructor is default. or should a similar error-message be issued for the copy-constructor too?

Subject: Re: Should the pick semantics be changed?
Posted by mirek on Wed, 05 Mar 2014 11:02:47 GMT
View Forum Message <> Reply to Message

piotr5 wrote on Wed, 05 March 2014 04:27
I agree, for now when compiling in c++11 the operator=(&) should produce a compilation error whenever instantiated, informing the programmer that picking is default and right side either must be enclosed in std::move (or maybe rather some customized Upp::Pick) or use operator<<= for deep copy.

I would not go that far as to use std::move. That part can stay as it is (I mean, '=' is transfer, either move or deep).

Mirek

Subject: Re: Should the pick semantics be changed?
Posted by mirek on Fri, 07 Mar 2014 08:00:50 GMT
View Forum Message <> Reply to Message

OK, I was playing a bit with &&:

```
#include <Core/Core.h>

using namespace Upp;

struct Foo {
 int a;

 Foo(Foo&& x)            { LOG("MOVE"); a = x.a; }
 void operator=(Foo&& x) { LOG("MOVE="); a = x.a; }

 Foo()       {}
 Foo(const Foo& x, int) { LOG("COPY"); a = x.a; }
};

template <class T>
T&& pick(T& a) { return static_cast<T&&>(a); }

template <class T>
T clone(const T& a) { T c(a, 1); return c; }

struct Bar {
 Foo a, b;

 Bar(Bar&&) = default;
 Bar& operator=(Bar&&) = default;
 Bar() = default;
 Bar(const Bar& b, int) : a(b.a, 1), b(b.b, 1) {}
};

Bar test_bar()
{
 LOG("In FN");
 Bar x, y;
 x.a.a = 123;
 y.a.a = 421;
 return Random(2) ? pick(x) : pick(y); // This is somewhat unexpected and ugly...
}

Foo test() {
 Foo h;
 h.a = 22;
 return h;
```

```
}

CONSOLE_APP_MAIN
{
 {
  Foo y;
  Foo z = pick(y);
  LOG("----------");
  Foo x = clone(z);
  LOG("----------");
  x = pick(y);
  LOG("----------");
  y = clone(x);
  LOG("----------");
  x = test();
 }
 LOG("======================");
 {
  Bar y;
  Bar z = pick(y);
  LOG("----------");
  Bar x = clone(z);
  LOG("----------");
  x = pick(y);
  LOG("----------");
  y = clone(x);
  LOG("----------");
  x = test_bar();
 }
}
```

I think this route is actually superior to what we have now, with all operations being stated directly where it matters.

Interestingly, 'clone' would work even with C++03 fine and is clearly superior to Foo x(y, 1); notation, perhaps even to operator<<=...

What sort of surprised me is the need to use 'pick' in test_bar. IMO, this makes that original purpose of && sort of moot...

Mirek

Subject: Re: Should the pick semantics be changed?
Posted by piotr5 on Fri, 07 Mar 2014 15:23:50 GMT
View Forum Message <> Reply to Message

I agree, imho this is a bug in gcc (or maybe also in c++11 standard): every return statement should pass the arguments as rvalue to the return-type constructor since the return statement is leaving scope and thereby its parameters have the same properties as all implicite rvalue objects: unlikely to remain (unless static), should not be altered except for purposes of picking, definitely a temporary value.

as for pick and clone I am unsure if we need both. this is the actual question of this thread: which of the two should we drop? in std-c++11 clone is implicitely done, always the default. in u++ the default is picking and clone would need to be stated. a good choice since picking is more efficient.

looking at this new idea with clone() I guess my fears about diverging from the standard has been soothed: if I want to use std::vector as drop-in replacement for Upp::Vector, I could specialize a clone() function for it. so I would be happy to see a clone function in Core, maybe already specialized for std library containers.

btw, I heard that according to the standard compilers are free to remove calls to the initializer of an object, so also from that side pick-construction can be realized. I haven't observed that behaviour though...

Subject: Re: Should the pick semantics be changed?
Posted by mirek on Fri, 07 Mar 2014 15:44:44 GMT
View Forum Message <> Reply to Message

piotr5 wrote on Fri, 07 March 2014 10:23I agree, imho this is a bug in gcc (or maybe also in c++11 standard): every return statement should pass the arguments as rvalue to the return-type constructor since the return statement is leaving scope and thereby its parameters have the same properties as all implicite rvalue objects: unlikely to remain (unless static), should not be altered except for purposes of picking, definitely a temporary value.

I am not so sure about this. I would say that what compiler requires here is a copy from lvalue to temporary object...

Quote:
as for pick and clone I am unsure if we need both. this is the actual question of this thread: which of the two should we drop? in std-c++11 clone is implicitely done, always the default. in u++ the default is picking and clone would need to be stated. a good choice since picking is more efficient.

Well, I got interested in this and after about 4 hours I have ide running with new notation. It is committed into branches/cpp11...

That being done, I would say that it is actually kind of nice to see all places where picking happens. Also, valid argument about unexpected behaviour (at least for beginners) ends with this.

Also, without mandatory 'pick', you need a lot of of constructors/operators= in the class. With 'pick', you need just

2 (or 3 if you want to support clone).

More good news: I am 90% sure that I can make clone and pick work in c++03: that would mean that we can have common sources for both, with the only difference that when compiled with C++11 active, you get things enforced.

Subject: Re: Should the pick semantics be changed?
Posted by mirek on Sun, 09 Mar 2014 08:34:32 GMT
View Forum Message <> Reply to Message

piotr5 wrote on Fri, 07 March 2014 10:23I agree, imho this is a bug in gcc (or maybe also in c++11 standard): every return statement should pass the arguments as rvalue to the return-type constructor since the return statement is leaving scope and thereby its parameters have the same properties as all implicite rvalue objects: unlikely to remain (unless static), should not be altered except for purposes of picking, definitely a temporary value.

thinking about it, consider

```
struct Foo {
  Vector<int> bar;

  Vector<int> Bar() { return bar; }
};
```

Now perhaps compiler could distinguish the situation where local lvalue is returned, but I guess rules would become complicated here....

Mirek

Subject: Re: Should the pick semantics be changed?
Posted by piotr5 on Sun, 09 Mar 2014 12:28:28 GMT
View Forum Message <> Reply to Message

well, to formalize a bit more concretely what I said, I believe return statement should at compile-time distinguish between return values local to the function which are not static, and all the return-values which remain in scope after the end of this function respectively could get into scope again (like static values or private members and such). and then based on this distinction the actual return-value should either be constructed with implicit cast to r-value or with the persistent const-l-value-reference constructor.

but then, I'm no it-scientist, so maybe someone else should better post this proposal to the c++

standard committee and to gcc -- if it hasn't already been proposed. I am quite certain gcc could easily implement such a language-change, without waiting for the standards-commitee's decision, I doubt it is explicitly disallowed in c++11...

oh, and thanks for the c++11 branch, looking for this was the reason I started this thread.

Subject: Re: Should the pick semantics be changed?
Posted by Lance on Sun, 23 Mar 2014 21:02:27 GMT
View Forum Message <> Reply to Message

piotr5 wrote on Sun, 09 March 2014 08:28well, to formalize a bit more concretely what I said, I believe return statement should at compile-time distinguish between return values local to the function which are not static, and all the return-values which remain in scope after the end of this function respectively could get into scope again (like static values or private members and such). and then based on this distinction the actual return-value should either be constructed with implicit cast to r-value or with the persistent const-l-value-reference constructor.

Not sure if I get you correctly but I think it's logical to put the burden on programmers as is. The two return types are different (and binary incompitable): one is an object and the other is a reference to object. Let's say C is a moveable class. There are some global/static objects of C which we can choose from one only from base on input parameter, then it's logical for the programmer the let the reference be the return type.

```
 /*const*/ C& getC(int type);
```

Now suppose we need to return some composed(not presently existing) C for some special input types(e.g., when type<100, there are corresponding existing global/static C objects, when type>=100, one has to be composed accordingly), the most economically way is to return a temporary C object if adding a new function to handle the case when type>=100 is not desirable.

```
C getC(int type);
```

Now if you declare the function prototype using the second way, any reasonable compiler would not be able to change it to return reference (the first) even if it's smart enough to detect that inside you implementation for getC actually involves only global/static C objects.

So a programmer has to make this decision, the compiler has to respect the programmer even if doing it otherwise is more efficient.

However, in the case a local object is returned, compiler may take the burden and save the programmer some extra key stroke., eg

```
C getC(int type)
{
```

```
    if( (unsigned)type<100 )
      return globalCs[type]; // return reference here is
                             // impossible as it will make
                             // the subsequent code impossible

    C c;
    createCBasedOnType(type, c);

    return c; // instead of
              // return std::move(c);
              // as required by current standard to take
              // advantage of the supposedly more efficient
              // move semantic
}
```

Subject: Re: Should the pick semantics be changed?
Posted by piotr5 on Tue, 25 Mar 2014 18:35:14 GMT
View Forum Message <> Reply to Message

you are almost correct on what the actual problem is, but your solution shows you think in terms of low-level languages. the goal of c++ is to abstract away the optimizations so that they can be handled seperately instead of polluting your code with it. in case of returning objects from a function this "promise" gets broken. u++ solved this problem by saying that all initializations of new objects happen by moving the contents of an object and thereby destroying the originating object. i.e. all containers in u++ are treated (on low-level) as if they were just references, while on high-level type-safety makes sure you don't use old objects that have been altered in the newly created container. now std-c++11 introduced something similar to what u++ has, but on the level of the compiler.

in your example, you would need to encapsulate the class C into another class which holds a const reference to it. then in getC you create new objects pointing at the C object you choose. this way you avoid copying C. additionally you have less need for whatever garbage collection since the old C object doesn't move around in memory, so you have no new fragmentation except for some gaps of pointer-size which will find some use quickly.

however, this thread talks about objects that are not constant. so you have this collection of static predefined objects, and you don't just return some constant stuff, you give ownership and control over C to a particular encapsulating object. this collection of static objects you had, now contains one object less, you already gave away exclusive control to someone for this removed object. inside memory nothing changed, the object you returned still occupies the same memory it always did, except for some overhead, it just isn't available anymore. for example if the compiler would generally place static objects inside of the same memory as the function's code (which is nonsense since code is usually read-only for the processor) then the returned object would work on that area instead on heap or stack or whatever.

you are wrong when you think my proposal would be about return-types, it's rather about object

initialization. there is an inconsistency in g++: when you write "new C(getC());" then on the heap the C object is initialized with r-value, it's syntactic sugar for "new C(move(getC));". however, if inside getC() you write "C o; createCBasedOnType(type,o); return o;" then this isn't syntactic sugar for "C o; createCBasedOnType(type,o); return move(o);". in the first case the syntactic sugar is rationalized by the impermanence of the C object getC() sends to the initializer. but also the object o which is the return-value of getC() is impermanent, it gets destroyed immediately after the return-value has been created. it is an inconsistency that both cases aren't treated the same in terms of syntactic sugar! it's not really a bug, but it's counterintuitive.

Subject: Re: Should the pick semantics be changed?
Posted by Lance on Wed, 26 Mar 2014 01:37:32 GMT
View Forum Message <> Reply to Message

Sorry, can you demonstrate your original proposal with some code?

I mean just explain what you mean by
Quote:
well, to formalize a bit more concretely what I said, I believe return statement should at compile-time distinguish between return values local to the function which are not static, and all the return-values which remain in scope after the end of this function respectively could get into scope again (like static values or private members and such). and then based on this distinction the actual return-value should either be constructed with implicit cast to r-value or with the persistent const-l-value-reference constructor.

If I didn't get you wrong, there is really little the compiler can do, or it's already what is being done.

Subject: Re: Should the pick semantics be changed?
Posted by Lance on Wed, 26 Mar 2014 01:48:11 GMT
View Forum Message <> Reply to Message

Quote:
there is an inconsistency in g++: when you write "new C(getC());" then on the heap the C object is initialized with r-value, it's syntactic sugar for "new C(move(getC));". however, if inside getC() you write "C o; createCBasedOnType(type,o); return o;" then this isn't syntactic sugar for "C o; createCBasedOnType(type,o); return move(o);". in the first case the syntactic sugar is rationalized by the impermanence of the C object getC() sends to the initializer. but also the object o which is the return-value of getC() is impermanent, it gets destroyed immediately after the return-value has been created. it is an inconsistency that both cases aren't treated the same in terms of syntactic sugar! it's not really a bug, but it's counterintuitive.

what g++ does is exactly what's required by the c++11 standard and hence will be done by compliant vc++ or other c++.

First case, getC() returns a tempoary (unnamed_, which is a r-value reference. while in the second case, o is named, std::move() utility or its equivalent is required to convert it into a r-value reference.

By the way, it has nothing to do with where it's created on the heap, it has to do with whether you are constructing with a temporary or a named a varaible, eg

```
struct C{

C(const C& c){}
C(C&& c){}

};

C getC()
{
   C c;
   return std::move(c);
}

int test()
{
   C c=getC(); // the move constructor will be called;
   C*p=new C(getC()); // same move constructor;

   C d=c; // copy ctor will be called;


   C e=std::move(c); // with named variable, std::move or
         // equivalent is required to invoke the move ctor
}
```

Subject: Re: Should the pick semantics be changed?
Posted by piotr5 on Wed, 26 Mar 2014 09:35:38 GMT
View Forum Message <> Reply to Message

Lance wrote on Wed, 26 March 2014 02:48it has to do with whether you are constructing with a temporary or a named a varaible, eg

```
struct C{

C(const C& c){}
C(C&& c){}
```

```
};

C getC()
{
   C c;
   return std::move(c);
}

int test()
{
   C c=getC(); // the move constructor will be called;
   C*p=new C(getC()); // same move constructor;

   C d=c; // copy ctor will be called;


   C e=std::move(c); // with named variable, std::move or
         // equivalent is required to invoke the move ctor
}
```

yes, that's the example I had in mind. here at the end of getC() the return statement turns c into a temporary since it's getting out of scope after that operation -- very much as in 1st and 2nd line of test(). I'm not saying that behaviour would or wouldn't go against c++11, I just doubt the standard is explicit enough to prevent other compilers from implementing what I said: distinguish between nonstatic local scope variables and all the rest, and just treat those variables which will be destroyed anyway as if they were temporaries, pass them by r-value to the returned class automatically. or can you quote anything from the standard which says that return-values must be initialized by copy-constructor in those cases? if there were a hundred implementations of the c++11 standard, I doubt all 100 would work that way. to convince me otherwise you'd have to quote the definition of what temporaries are and how they differ from values which will be destroyed immediately after this particular statement.

the programmer can't do anything about it, imho changing such behaviour wont destroy compatibility to existing programs, maybe it would improve their performance. and it would definitely remove the burden of optimizing the sourcecode from some of the programmers. programmers wouldn't need to go through their code, searching for all the return statements, and enclosing the returned objects into "move()" wherever it makes sense! the compiler should take over that optimization task! if gcc is a bit frightened about that change, maybe it should be a compiler-option. but it definitely is needed since programmers cannot influence that except by using automatic code-improvement tools, or code-analysis tools which nag them about some forgotten move() in a return statement...

Subject: Re: Should the pick semantics be changed?
Posted by mirek on Wed, 26 Mar 2014 10:38:05 GMT
View Forum Message <> Reply to Message

One way or another, I believe that explicit 'clone' seems like quite a good idea....

Subject: Re: Should the pick semantics be changed?
Posted by Lance on Wed, 26 Mar 2014 20:21:00 GMT
View Forum Message <> Reply to Message

Oh I see. That I have no problem. Actually at the end of my first post in this thread I mentioned the same(in the sample code). This indeed is a case where compiler writers could take the burden and save programmers a few keystrokes if so required by the standard.

Subject: Re: Should the pick semantics be changed?
Posted by Lance on Wed, 26 Mar 2014 20:25:39 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 26 March 2014 06:38One way or another, I believe that explicit 'clone' seems like quite a good idea....

Yap. Make sure the programmer is aware of what he/she is doing and are ready to pay the price by removing a default.