Subject: Re: Few questions
Posted by OvermindDL1 on Sat, 06 Nov 2010 01:04:40 GMT
View Forum Message <> Reply to Message

mr_ped wrote on Fri, 05 November 2010 02:41OvermindDL1 wrote on Fri, 05 November 2010 05:48 - Is there any Boost package, preferably to the trunk?

No. Does it even need a package, isn't it just set of headers to include? (I don't use Boost, so I'm not sure)

About creating package... simply create package, include the .cpp/.h/other project files (.cpp inclusion is crucial for project build, .h and other directly non-build-able files just for easy edit trough IDE).
Then add the package to whatever other package where you want to use it.

If you refresh the sources of that boost package trough SVN, build process should detect changes in "modified" timestamp and rebuild everything related. (although with such fundamental classes used all over place it's more safe to hit Rebuild All, especially if you get some weird error after ordinary build)

I'm very likely wrong, but as long as Boost is just a helpful .h header, I don't see any problem with building it and using it in U++ (without any modification to source).

Boost is a meta-library, not a library in and of itself, but rather a lot of smaller libraries (some *very* small, some quite large) just held together under one umbrella, if they are good enough. As such, most are header-only, some require pre-compilation (using the bjam build engine, although a CMake port was finally 'completed' after much effort, bjam was made to work around the multi-platform inconsistencies very well), and some have a header-only or build option depending on if you want higher optimization or faster link time, and the build ones all (sans like two of them that have to be shared by their design) can be shared libraries or static libraries.

Reason I ask is that Boost is not auto-picked up from my global MSVC headers it seems, so I either need to explicitly include its directory in every one of my projects (hard-coding in the directory is *not* good in my opinion), or need to add the lookup to the U++ build methods, hence a package.

mr_ped wrote on Fri, 05 November 2010 02:41Quote:
- Is U++ able to update the source of a project automatically using svn checkout/export (does it come with an svn client in other words that I can call from script?).

TheIDE has SVN support in menu, it does use common command line client (svn.exe in win and svn in POSIX), which is IIRC not part of U++ installation. So TheIDE is just an front-end for classic SVN client, nothing more. (and if you prefer to have more control about what's happening under hood, I would suggest to use external tool like TortoiseSVN, TheIDE is a bit too simplified for my taste)

That would be sufficient for my use, just some easier way to update from boost trunk instead of

needing to do it manually, although that could be done too.

mr_ped wrote on Fri, 05 November 2010 02:41Quote:
 - If I do port boost/trunk to a reusable UPP package, would anyone be interested in it in the bazaar?

 Me: not now, nor planning it. But it never hurts to have something working in bazaar, so in case the need arise, you can try it out in ~3 clicks of mouse.

It is better then hardcoding in include links for sure, I am still very surprised that this was not already done considering how massive and useful Boost is.

mr_ped wrote on Fri, 05 November 2010 02:41Quote:
 - Would also think of making a boost/upp package as well to include headers to let Boost work with many UPP structures, one of the nice things about boost is its extensibility to support other container, structure, etc... types, would this be welcome?

 This requires some work and R&D and experiments, right?
Of course anything of this is welcome and highly appreciated, just make sure your effort is well documented, so you will save time of other U++ users and make a path for them.
 (If your idea of "welcome" is that it will get heavily adopted soon... take a break, this is BSD world, you give things away and see what picks up, but don't hold your breath.)

I know how the world is, I shun the limited gpl like a plague, I only release under truly free licenses, such as Boost (even BSD is 'less-free' then the Boost license, but I still use it).

mr_ped wrote on Fri, 05 November 2010 02:41Quote:
 - Hard to get an indication from these forums (the bazaar forum appears to have even fewer submissions then Boost, and it is *really* hard to get a submission into Boost...), but how active is the community and so forth?

 Community is small, but very active. Getting submission into bazaar is not very hard (even of lower quality or work in progress), getting patch into uppsrc is reasonably (much more) difficult (90% of good patches get in, and 90% of ballast get rejected, I don't think you can do much better with project of this size).
 Bazaar pretty much follows it's name. If you dare to share your work, you will very likely get it there.

Heh, so no multi-year-long review process then like I am used to then?

mr_ped wrote on Fri, 05 November 2010 02:41Quote:
 - Is there a good third-party package repository of various libraries 'ported' to the U++ package format for ease of use U++ projects?

 Bazaar is best place to start, then this forum. I'm not aware of anything for U++ being advertised elsewhere and not here, so I guess the answer is "no".

So the amount of packages is relatively low overall then? I shall work on fixing that as I get

needs.

mr_ped wrote on Fri, 05 November 2010 02:41Quote:
- Is there a mailing list, IRC, newsgroup, etc..., or just these forums?

Forums are fastest and most active channel. There were also some IRC discussions, and also already a small live session in Prague, then PM on forums and some IM, but basically if you are active on forum, you shouldn't miss anything.

Well I am stuck here in the USA, so a bit far for me. Forums are fine for me, just more used to mailing lists due to their ease of use and power.

mr_ped wrote on Fri, 05 November 2010 02:41Quote:
My background, been programming in C++ for going on 18 years, huge advocate of Boost, regardless of its monolithicly huge compile times in certain libraries (Spirit is such a great parser...), and I have a tendency to push things to their limits. Generally program in Visual Studio with the Visual Assist plugin for Windows, *nix, and Mac programming work (Virtual Machines are great) due to how amazingly productive it is, Visual Assist practically reads my mind it seems, testing out U++ + TheIDE to see if it will make a better 'native' multi-platform build engine and IDE, and as stated above, I am impressed with the layout of how builds work, tag based systems can be very nice.

Nice to have you here, you can surely contribute by feedback of your U++ experience (as it's usually quite a rough ride at the beginning) and I hope the community will be able to help you. Although with your experience and being a avid boost user you are not truly "compatible" with U++, so don't be surprised if it takes some time for you until you find a good way how to exploit U++ for your advantage. You will have to relearn some things for no obvious gain, and some things you will very likely do more efficiently in MSVS+boost, if you are good at it. Still if you ask me, U++ is worth a try, especially if you like to push things on limit, there are areas where U++ is clear winner.

Only been messing with it for a day now, I have not seen any rough ride, it seems quite smooth to me. The 'pick'ing, which I just take as 'move'ing in modern C++ terminology, has always made a lot of sense to me and it seems perfect for me.

I do question one design aspect though, why use an unused integer as a differentiator between pick/move and copy constructors and so forth? To me it seems better to use one of these two designs:

```
// instead of this:
  myType t(getMyType); // pick/move
  myType t(getMyType, 1); // copy

// Should do one of these for copy:
  myType t(copy(getMyType));
    // possibly renamed from copy,
    // but a single argument, copy should do nothing but return
```

```
    // something like _copy<myType> that just contains a
    // reference, yes, reference to MyType, which would call a
    // constructor specialized on that _copy<myType> type or so.

// Or this way:
  myType t(getmyTpe(), COPY);
    // where COPY is just a struct _COPY{}; static _COPY COPY;
    // or so, empty, etc...
```
The advantage of both of those is less register pressure in the call by not including an integer as an argument, and no chance of ambiguity if the call (potentially not a constructor call but a function call) if more parameters are passed.

The second method is more like your current method but reduces register pressure and removes ambiguity.

The first method, however, can also let the type have a conversion to the _COPY container as well so a function like void callMe(_COPY<myType> t); would work when myType t;callMe(t); is called so the function would always make a copy and never move without needing to add scaffolding inside of it to make an explicit copy after taking myType by reference or so, so that saves a line of code, *plus* it documents the method signature.

Also note, Boost *loves* move semantics, so it seems that U++ would fit right transparently in after making adapters to integrate them. There is actually a reviewed library (which I need to finish reviewing... >.<) that adds a C++1x style move semantic library including replacement std containers that support them, basically like U++ containers, but to move something you have to actually do something like:
mytype t(move(getMyType()));
Else it defaults to copy as per the C++1x standard.

Even if I do not use TheIDE, the U++ library itself seems quite useful so I would probably end up using it anyway, the gui aspect seems quite useful for some simple projects of mine without needing to whip out all of wxWidgets.

mr_ped wrote on Fri, 05 November 2010 02:41Quote: - Have you though about including clang++ as a build engine? You could even import its sema library to provide full VisualAssist-level and above code verification, analysis, intellisense, etc...?
clang in latest version is capable to build TheIDE, dolik is toying with it. Basically the building with clang already works, search forum to see what additional setup is needed. (you need working GCC build method and then just add clang build method as an replacement variant to GCC build)

It would be nice to include clang with TheIDE including a set of headers for pure multi-platform use thus not requiring a previous setup and not having to worry about compiler differences on various platforms

mr_ped wrote on Fri, 05 November 2010 02:41And, one more thing about additions to Bazaar: BSD/MIT license or compatible is a good start for new package. (From reading the boost license right now I would say it's BSD compatible, but I didn't dig deep into it) If it's not compatible, you should mark it clearly as such (or maybe not include it into bazaar at all and push it just to

forums), because most of the U++ users expect everything in bazaar to be free to use in commercial apps without consequences.

The Boost license is compatible with BSD, and it is in fact even more free then BSD as it does not require an text saying that you are using or anything, about the only thing it says is there is no warranty and you cannot claim that it is still boost code if you make modification, thus, if you make a modification to a boost header you cannot claim that it is still boost code, but rather your own, and you are allowed to change the license to pretty much whatever you wish. Boost can be included with any project, no charge, etc...

dolik.rce wrote on Fri, 05 November 2010 03:51Hi Overmind,

Welcome to the forum Mr_ped pretty much answered all you questions, so just a few details:

There is IRC channel #upp at slashnet.org. I am online most of the time, other than me only about 2 people drop by occasionally.

I might hover around in there, not been on that server before so would need to setup another account first.

dolik.rce wrote on Fri, 05 November 2010 03:51Clang works fine on linux and probably other unix-like systems. There are still some problems on windows.

Actually Windows support on it works quite well for most people now, assuming you use mingw's headers and not Visual Studio's, although support for Visual Studio's headers is increasing at a very rapid pace.

dolik.rce wrote on Fri, 05 November 2010 03:51As for deeper integration (inellisense, code analysis,...) it would require bit more work (especially the intellisense, as it would have to act as a replacement to Assist++, which is part of theide). But I'm not saying it won't happen at some point in future.

Correct, Assist++ would be rewritten, but you would have absolute and full parse, semantic, etc... information of the source code in question, even being able to follow back through complicated template or even macro expressions.

dolik.rce wrote on Fri, 05 November 2010 03:51U++ is very good in pushing the limits. Not only the limits of what can be done in C++, but often also the limits of programmer It learned me huge amount of new things about C++ and programing in general and also a lot of useful tricks... It is definitely worthed a try

I am curious, if you had to sum up the limits that U++ pushes other then pick'ing/moving support, what else would you say that it is? To me it seems like a well made library, but does not particularly push much. Although, to be honest, my definition of 'pushing' something is like Boost.Spirit (assuming U++ is properly exposed to Boost):

```
upp::Vector<int> vi;
```

```
upp::String s("1,2,3,42");
boost::spirit::parse(start(s), end(s), int_%',', vi);
// vi now contains {1, 2, 3, 42}
// And spirit expressions can be complex enough that there is
// even one that can parse a C-style language and more.
```

Or like Boost.Phoenix (assuming U++ is properly exposed to Boost):

```
template<typename cbType> void mapInts(Vector<int> &v, cbType cb)
{ // assuming foreach exists
  foreach(int &i, v) cb(i);
}

// Hmm, lets make one for something else too
struct myType {int i; float f;} // assume this is properly exposed to Boost too
typename boost::fusion::vector<int,float> myTuple;

template<typename T, typename cbType> void map(Vector<T> &v, cbType cb)
{ // assuming foreach exists
  foreach(T &i, v) cb(i);
}

// And how to use Phoenix in a situation like this,
// and Phoenix is *very* generically useful:
Vector<int> v = {1,2,3,4,5}; // assuming this supports C++1x init lists for ease of reading
Vector<int> v1(v,1); // copy
Vector<int> v2(v,1); // copy
Vector<int> v3(v,1); // copy
mapInts(v1, arg1*=2);
mapInts(v2, if_(!(arg1%2))[arg1/=2].else[arg1=0]);
mapInts(v3, let(_a=arg1)[
            while_(--arg1)[
              _a+=arg1
            ]
            ,arg1=_a
          ]);
assert(v1 == {2,4,6,8,10});
assert(v2 == {0,1,0,2,0};
assert(v3 == {1,3,6,10,15};

myType ty(42,3.14);
myTuple tu(2,3.4);
map(ty, arg1*=2);
map(tu, arg1/=2);
assert(ty == myType(84, 6.28)); // assuming the floats compare equally
assert(tu == myTuple(1, 1.95)); // assuming the floats compare equally
```

Basically, phoenix implements C++ in C++ lazily so you can pass a phoenix-c++ expression as a

functor, so you can do something like this:

```
std::function<void(int&)> f1(arg1*=2);
std::function<int(int)> f2(arg1*2);
std::function<int(int,float,std::string)> f3(((COUT()<<arg3<<arg2<<arg1<<"\n"),arg1*arg2));

auto myFun = arg1*2;
std::function<int(int)> f4(myFun);
std::function<float(float)> f5(myFun);

int i=2;
f1(i); // `i` become 4
f2(3); // returns 6
f3(2,3.5, "hi!"); // prints "hi!3.52\n" to COUT(), then returns 2*3.5, which
                  // is promoted to 2.0*3.5, which is 7.0, which is returned as int 7
f4(4); // return 8
f5(5.5); // returns 11.0

struct myType { int i; void operator*(int i){COUT()<< "i=" << i << "\n";}};
myType t;

myFun(3); // returns 6
myFun(3.3); // returns 6.6
myFun(string("hi")); // returns "hihi" as a string is the string type support repeating
                     // by multiplated amount
myFun(t); // prints "i=2", returns void
int j=myFun(t); // does not compile because myFun returns void here
int k=myFun(2); // k is set to 4
```

Unlike C++1x lambdas, which are monomorphic, boost::phoenix is polymorphic, meaning that it adjusts itself to the passed-in types. f1, f2, and f3 are just monomorphic in use, the polymorphism is examplified by myFun, f4, and f5. myFun is an unknown-type, meaning that if you want to save it to a variable you have to use either BOOST_AUTO or C++1x's auto keyword to save it to a variable, at this point it contains no data, is zero in size, and just an ast of types is generated (see Boost.Proto). When it is finally 'called', either by explicitly calling it like myFun(2), or by saving it to something that is capable of holding a functor (such as a boost/std::function, maybe U++'s callback methods depending on their power), then it generates code, the code is specific based on the input arguments and due to some template work there are no real functions generated, but is rather generated inline, thus doing myFun(2) will literally generate 2*2 inline, which the optimizer optimizes to an inline 4, hence no code is generated other a constant 4, and myFun(i) literally compiles inline to i*2, and is optimized by the optimizer if possible, all at compile time. If saved to an std::function<int(int)> for example, a single functor is generated that contains the highly optimized code. In the much above example mapInts function, the cbtype is a function parameter and not a boost/std::function as you may have noticed, meaning the real phoenix type is passed in, which is not a functor yet, but is then called in the loop, which inlines the call in the loop, meaning that the loop contains no function call, it is all done right there, very fast, all at compile time.

Admittedly, things like Spirit and Phoenix, which are *heavy* in modifying the C++ AST by using templates (again, see Boost.Proto, which is how they are implemented), do cause greatly increased compile time, but they both give you greatly increase runtime performance. The characteristics of Phoenix let you create lazy C++ function inline without needing to create external functions/functors to call, which may not be inlined like phoenix will be. The characteristics of Spirit generate parsing code (actually Spirit.Qi creates parsers from streams, usually text, but could be binary, to C++ structures, and Spirit.Karma goes the other way, from C++ structures to a stream, and Spirit.Lex is a lexer for ease of use for some patterns for parsing) at compile time that use both embedded optimizers to optimize the C++ AST using templates, then is further optimized by using the C++ optimizer in the compiler itself, as such, Boost.Spirit is currently the fastest parsing engine on the planet, outperforms every single tested parsing engine from Yacc/bison/antlr/hand-coded/etc, and so much more. If there is ever a case where Spirit is outperformed by something else, a simple test case sent to the devs can let them specialize a template optimizer to transform the code into something better for those use-cases, as such, Spirit can and always shall perform better then any other parsing engine.

As a disclaimer, yes, I have helped create both Phoenix and Spirit in small ways, so I do like to show them off. ^.^

So yes, those are what I traditionally think of when I think of 'pushing' C++.

As for pushing the programmer, I know of the move semantics, they are getting popular in optimized C++ applications, and I see how the gui engine is designed, minimizing pointers, etc..., but it is all quite clear to me, so what are you describing about pushing the programmer? I have made continuation and coroutine engines in C++, made an Actor style library to all but remove all locks in multi-threaded programming to allow arbitrary expansion across any amount of cores, so I am used to 'pushing' things.
