Subject: NEW: generic Toupel grouper
Posted by kohait00 on Thu, 12 Aug 2010 14:03:51 GMT
View Forum Message <> Reply to Message

sometimes its cool to group things together in an easy manner, especially when wroking with the template containers. to build a grouping container over and over again each time for different purposes, just for classes that would have the parts public anyway,is odd.

say you want to have a vector that simply contains a 2-tupel of values. maybe some measurements consisting of two values at once, or anything that logically belongs together but does not need any abstraction or scope hiding. a normal container cant do it.

Vector<float, float> vi; //would resemble VectorMap.

you just need to quickly define and access 2 things at once in a container and want to have sth like

```
Vector<Duo<int, int> > vi;
...
vi[i].t1 = 123;
vi[i].t2 = 234;
```

this looks good and is handy.

so here comes some helpers to do that

```
template<class T>
class Solo
{
public:
 typedef Solo<T> CLASSNAME;
 Solo(const T & _t) : t(_t) {}
 Solo() {}

 operator T & () { return t; }
 operator const T & () const { return t; }

 T t;
};

template<class T1, class T2>
class Duo
{
public:
 typedef Duo<T1, T2> CLASSNAME;
```

```
 Duo(const T1 & _t1, const T2 & _t2) : t1(_t1), t2(_t2) {}
 Duo() {}

 operator T1 & () { return t1; }
 operator const T1 & () const { return t1; }
 operator T2 & () { return t2; }
 operator const T2 & () const { return t2; }

 T1 t1;
 T2 t2;
};

template<class T1, class T2, class T3>
class Trio
{
public:
 typedef Trio<T1, T2, T3> CLASSNAME;
 Trio(const T1 & _t1, const T2 & _t2, const T3 & _t3) : t1(_t1), t2(_t2), t3(_t3) {}
 Trio() {}

 operator T1 & () { return t1; }
 operator const T1 & () const { return t1; }
 operator T2 & () { return t2; }
 operator const T2 & () const { return t2; }
 operator T3 & () { return t3; }
 operator const T3 & () const { return t3; }

 T1 t1;
 T2 t2;
 T3 t3;
};

template<class T1, class T2, class T3, class T4>
class Quartett
{
public:
 typedef Quartett<T1, T2, T3, T4> CLASSNAME;
 Quartett(const T1 & _t1, const T2 & _t2, const T3 & _t3, const T4 & _t4) : t1(_t1), t2(_t2), t3(_t3),
t4(_t4) {}
 Quartett() {}

 operator T1 & () { return t1; }
 operator const T1 & () const { return t1; }
 operator T2 & () { return t2; }
 operator const T2 & () const { return t2; }
 operator T3 & () { return t3; }
 operator const T3 & () const { return t3; }
 operator T4 & () { return t4; }
```

```
 operator const T4 & () const { return t4; }

 T1 t1;
 T2 t2;
 T3 t3;
 T4 t4;
};
```

maybe they can go to Others.h..

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Thu, 12 Aug 2010 14:07:16 GMT
View Forum Message <> Reply to Message

or, which would be better maybe, to have Two<T1, T2>, Three<T1,T2,T3>, Four<T1,T2,T3,T4> as analoge to One<T>..

but sometimes that much of complexity is not needed

Subject: Re: NEW: generic Toupel grouper
Posted by koldo on Thu, 12 Aug 2010 14:14:33 GMT
View Forum Message <> Reply to Message

Hello Kohait

This does not seem bad. However is it worthwhile the added code to support it?, as we always can insert in a Vector any class or struct.

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Thu, 12 Aug 2010 14:58:51 GMT
View Forum Message <> Reply to Message

ofcorse..but imagine, you need to setup a new class each time you simply just want to group/pack some things together without further class implications / namespaces / accessscopes. i imagine this to be quite often the case. (Point_ is not quite the same but is a small example of grouping things)

but here comes another option, which is maybe better...donnow.

```
template<class T>
class O
{
```

```
public:
 typedef O<T> CLASSNAME;
 O(const T & _t) : t(_t) {}
 O() {}

 operator T & () { return t; }
 operator const T & () const { return t; }

 T t;
};

template<class T>
class O1 : public O<T> {};

template<class T1, class T2>
class O2 : public O<T1>, public O<T2> {};

template<class T1, class T2, class T3>
class O3 : public O<T1>, public O<T2>, public O<T3> {};

template<class T1, class T2, class T3, class T4>
class O4 : public O<T1>, public O<T2> , public O<T3> , public O<T4> {};
```

beeing able to access stuff like this, which is more clear

```
 O2<int, float> o2;

 o2.O<int>::t = 123;
 o2.O<float>::t = 23.10f;
```

Subject: Re: NEW: generic Toupel grouper
Posted by koldo on Thu, 12 Aug 2010 15:03:18 GMT
View Forum Message <> Reply to Message

->

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Thu, 12 Aug 2010 15:04:33 GMT
View Forum Message <> Reply to Message

even got another idea..

```
template<class T1, class T2>
class Two : public One<T1>, public One<T2> {};

template<class T1, class T2, class T3>
class Three : public One<T1>, public One<T2>, public One<T3> {};

template<class T1, class T2, class T3, class T4>
class Four : public One<T1>, public One<T2>, public One<T3>, public One<T4> {};
```

EDIT: too quick forgot a type

```
 Four<int, char, float, unsigned> four;
 four.One<int>::Create() = 123;
```

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Thu, 12 Aug 2010 15:27:40 GMT
View Forum Message <> Reply to Message

a bit more usable

```
template<class T>
class O
{
public:
 typedef O<T> CLASSNAME;
 O(const T & _t) : t(_t) {}
 O() {}

 operator T & () { return t; }
 operator const T & () const { return t; }

 T& operator=(const T & _t) { t = _t; return t; }
 bool operator ==(const T& _t) const { return (t==_t); }

 T t;
};
```

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Thu, 12 Aug 2010 18:14:51 GMT
View Forum Message <> Reply to Message

Hi kohait00,

I've got two questions

1) Does it work when you need to group several values of same type? E.g. Three<String,int,int>. I have a suspicion that the overloaded operators for int wouldn't make much sense than...

2) What is the advantage of your touples to using simple Vector<Value>? The later might have bit more verbose interface, but on the other hand it offers variable element count.

Best regards,
Honza

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Thu, 12 Aug 2010 18:58:12 GMT
View Forum Message <> Reply to Message

hey doli, you are absolutely right with your point 1)
but in this case Duo<int,int> work, becasue there the items are named explicitely.

de advantage is, that zou dont always want to store all data inside Value, but as an explicit tzype.

but using same tzpe one might think of using Duo<String, Vector<int> >, grouping it

it is still a development, so expect some changes here. but the idea is to loosely group stuff together without the need to create a whole new class for it.. its maybe considred a class template, instead of a template class. a public members class.

cheers

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Fri, 13 Aug 2010 08:53:29 GMT
View Forum Message <> Reply to Message

Actually, I think this feature is somewhat missing in U++ (and is quite common in other frameworks).

But I am quite sceptical about those cast overloads too.

Imo, something as simple as

```
template<class T1, class T2>
struct Pair {
 Pair(const T1 & _t1, const T2 & _t2) : t1(_t1), t2(_t2) {}
```

```
 Pair() {}

 T1 first;
 T2 second;
};
```

would solve most of usage scenarios where I was eventually missing tuples.

Maybe first/second are bad names though...

Hm, maybe, what about

```
template<class T1, class T2>
struct AB {
 AB(const T1 & _t1, const T2 & _t2) : t1(_t1), t2(_t2) {}
 AB() {}

 T1 a;
 T2 b;
};
```

?

Here is boost's take on the issue:

http://www.boost.org/doc/libs/1_34_0/libs/tuple/doc/tuple_us ers_guide.html

IMO a little bit overengineered...

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Fri, 13 Aug 2010 08:58:24 GMT
View Forum Message <> Reply to Message

simple is most times better

good option. maybe AB, ABC is not 'verbose' enough (as of readability of code, but tfcorse a short option)

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Fri, 13 Aug 2010 09:16:28 GMT
View Forum Message <> Reply to Message

Quote:
But I am quite sceptical about those cast overloads too.

its just for easy handling, while opening doors to implicit casted assigns to wrong parameter (dont know if compiler would warn in such case, which operator() to take)

can be left out i think. if user uses it, he can type some 2 more letters and be on safe side. if he really needs this behaviour, polimophism is a good thing..

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Fri, 13 Aug 2010 12:59:10 GMT
View Forum Message <> Reply to Message

i found AB, ABC etc. beeing difficult to use, because wingdi.h already defines ABC.

ive looked into the boost stuff, it's really a bit over the line.
beeing on the secure side and type one letter is cool enough..

so this is a proposal, short enough not to blow Others.h

```
template<class T1, class T2>
class Duo
{
public:
 Duo(const T1 & _a, const T2 & _b) : a(_a), b(_b) {}
 Duo() {}
 T1 a; T2 b;
};

template<class T1, class T2, class T3>
class Trio
{
public:
 Trio(const T1 & _a, const T2 & _b, const T3 & _c) : a(_a), b(_b), c(_c) {}
 Trio() {}
 T1 a; T2 b; T3 c;
};

template<class T1, class T2, class T3, class T4>
class Quartett
{
public:
 Quartett(const T1 & _a, const T2 & _b, const T3 & _c, const T4 & _d) : a(_a), b(_b), c(_c), d(_d)
{}
 Quartett() {}
 T1 a; T2 b; T3 c; T4 d;
```

```
};
```

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Fri, 13 Aug 2010 18:21:28 GMT
View Forum Message <> Reply to Message

The boost implementation is really overkill. But one thing I like about it is the "indexed" access using the get<N>() function.

Also, something like //for Two (similar for bigger touples):

```
 Value operator[](int i)const{
  ASSERT(i>=0&&i<2);
  if(i==0) return a;
  else return b;
 } would be nice thing to have.
```

Honza

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Fri, 13 Aug 2010 23:40:14 GMT
View Forum Message <> Reply to Message

So, after criticizing kohaits proposals I felt obligated to show my own idea about how Touples should be implemented... Here is the result: #include <Core/Core.h>
using namespace Upp;

```
String AsString(Nuller n){return "";}

template <int N,class A,class B,class C> struct TTypes         {typedef Nuller Type;};
template <class A,class B,class C>       struct TTypes<0,A,B,C> {typedef A Type;};
template <class A,class B,class C>       struct TTypes<1,A,B,C> {typedef B Type;};
template <class A,class B,class C>       struct TTypes<2,A,B,C> {typedef C Type;};

template<class A, class B,class C> struct Touple;

template<int N,class A,class B,class C> struct Retriever{
 static Nuller Get(Touple<A,B,C>& t){return Null;};
};
template<class A,class B,class C> struct Retriever<0,A,B,C>{
 static typename TTypes<0,A,B,C>::Type Get(Touple<A,B,C>& t){return t.a;};
};
template<class A,class B,class C> struct Retriever<1,A,B,C>{
 static typename TTypes<1,A,B,C>::Type Get(Touple<A,B,C>& t){return t.b;};
};
```

```
template<class A,class B,class C> struct Retriever<2,A,B,C>{
 static typename TTypes<2,A,B,C>::Type Get(Touple<A,B,C>& t){return t.c;};
};

template<class A, class B=Nuller,class C=Nuller>
struct Touple{
 A a;
 B b;
 C c;
 Touple(){};
 Touple(A a):a(a){};
 Touple(A a,B b):a(a),b(b){};
 Touple(A a,B b,C c):a(a),b(b),c(c){};

 template<int N>
 typename TTypes<N,A,B,C>::Type Get(){
  return Retriever<N,A,B,C>::Get(*this);
 };
 template<class T>
 Touple& operator=(const T& t){
  a=t.a; b=t.b; c=t.c;
 }
 int GetCount()const{
  for(int i=2; i>0; i--){
   if((*this)[i]!=Value(Null)) return i+1;
  }
  return 1;
 }
 Value operator[](int i)const{
  if     (i==0) return Value(a);
  else if(i==1) return Value(b);
  else if(i==2) return Value(c);
  else ASSERT_(false,"index out of bounds");
 }
 String ToString()const{
  String s=AsString(c);
  s=AsString(b)+(s.GetLength()>0?",":"")+s;
  s=AsString(a)+(s.GetLength()>0?",":"")+s;
  return "{"+s+"}";
 }
};

template<class A>
Touple<A> Solo(const A& a){
 return Touple<A>(a);
};

template<class A,class B>
```

```
Touple<A,B> Duo(const A& a,const B& b){
 return Touple<A,B>(a,b);
};

template<class A,class B,class C>
Touple<A,B,C> Trio(const A& a,const B& b,const C& c){
 return Touple<A,B,C>(a,b,c);
};

CONSOLE_APP_MAIN{
 Touple<double,const char*> s;
 Touple<int,String> t;
 Touple<int,String,double> u;

 t.a=1; t.b="test";
 DUMP(t.Get<0>()); DUMP(t.Get<1>());
 for(int i=0; i<t.GetCount(); i++){
  LOG(i<<": "<<t[i]);
 }
 s=t;                          DUMP(s);
 t=Duo(1,String("hello world"));  DUMP(t);
 u=t;                          DUMP(u);
 u.c=3.2;                        DUMP(u);
 // <double>=<triple> fails to compile:
 // t=Triple(1,String("dsd"),3);  DUMP(t);
}
```

The main difference is that there is no specific type for two,three, etc. values, but rather a single Touple class that can be used universally. The implementation above allows 1,2 or 3 elements, but can be easily extended. The main idea is that smaller touple can be assigned into bigger (extra elements are Null), while bigger into smaller triggers compilation errors. The functions Solo,Duo and Trio are supposed to save you some typing by generating the touples based on the arguments types. The only thing I found missing in Core was AsString(Nuller) which should be no problem to add.

The ideas from my previous post are included. I am aware that touple.Get<0>() doesn't have any significant syntactic value, since it is the same as touple.a, but it might help the readability a bit. On the other hand, the operator[] is quite important, since it allows iterating through the Touple. TEven though it returns Values, together with GetCount() (returning number of elements from 0 to last non-Null) should be quite powerfull tool.

What do you think?

Honza

PS: The Solo,Duo and Trio names were chosen just because there already is One and Single. Otherwise I would prefer Single, Double, etc.

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Sat, 14 Aug 2010 09:53:43 GMT
View Forum Message <> Reply to Message

dolik.rce wrote on Fri, 13 August 2010 14:21The boost implementation is really overkill. But one thing I like about it is the "indexed" access using the get<N>() function.

Also, something like //for Two (similar for bigger touples):
```
 Value operator[](int i)const{
  ASSERT(i>=0&&i<2);
  if(i==0) return a;
  else return b;
 } would be nice thing to have.
```

Honza

You expect too much about types involved here IMO.

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Sat, 14 Aug 2010 10:25:23 GMT
View Forum Message <> Reply to Message

luzr wrote on Sat, 14 August 2010 11:53dolik.rce wrote on Fri, 13 August 2010 14:21The boost implementation is really overkill. But one thing I like about it is the "indexed" access using the get<N>() function.

Also, something like //for Two (similar for bigger touples):
```
 Value operator[](int i)const{
  ASSERT(i>=0&&i<2);
  if(i==0) return a;
  else return b;
 } would be nice thing to have.
```

Honza

You expect too much about types involved here IMO.
I am aware of that. But if I am not mistaken, it doesn't affect functionality of the class too much. The compilation should fail only in case when the operator is actually used, since it is templated. If user knows the type is not compatible, he will just have to resort back to using the members directly. There should be no or minimal performance penalty for that... Am I right?

Moreover, there are some further assumption made anyway, about compatibility with Null. That might be even more restricting...

Honza

Quote:
Otherwise I would prefer Single, Double, etc.

there, similarities to data types wouled be to close (Double) and misleading. Solo is not usefull anyway (wraping one T is .. nonsense), it should start with Duo, Trio, etc.. to set Apart from the One (and maybe later Two, Three)

Quote:
after criticizing kohaits proposals

dont worry, it's part of development, and didnt feel criticised . i am always looking forward to meeting better ideas..

to the above implementation:

the Get<1>() option (idea from boost?) is a cool trick, but IMHO of little use because it's not a runtime check, but a compile time definition, thus u.a, u.b, u.c is much simpler and clearer in that sense, and less to type anyway. i mean, in terms of compile time specialisation u.Get<1> is same as u.a, you have to provide the index at compiletime, so you know which type.

the Value operator[](int i) is a good idea though. to wrap / unwrap in value (boxing / unboxing is used in C# and others, though there in different context, as base class object).

having Duo, Trio, etc is, as you pointed out, more or less useless, even if it's better to read so i added a 5th T and deaulted past second T. (i'd rater use EmptyClass, but there is no Value(const EmptyClass &) for it, so i used Nuller. might be usefull to have an EmptyClass Value as well?)

so here comes another option.

```
template<class T1, class T2, class T3=Nuller, class T4=Nuller, class T5=Nuller>
class Tupel
{
public:
 Tupel(const T1 & _a, const T2 & _b, const T3 & _c, const T4 & _d, const T5 & _e)
  : a(_a), b(_b), c(_c), d(_d), e(_e) {}
 Tupel() {}

 Value operator[](int i) {
  switch(i) {
   case 1: return Value(a);
   case 2: return Value(b);
   case 3: return Value(c);
   case 4: return Value(d);
   case 5: return Value(e);
```

```
  default:
  case 0: ASSERT(0); return Value();
 }
 return Value(); //dummy
}

 T1 a; T2 b; T3 c; T4 d; T5 e;
};
```

what about this one? it provides the simplicity desired and has the wrapper.

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Sun, 15 Aug 2010 11:09:21 GMT
View Forum Message <> Reply to Message

Hi Kohait

kohait00 wrote on Sun, 15 August 2010 09:16dont worry, it's part of development, and didnt feel criticised . i am always looking forward to meeting better ideas..
I don't worry, it was a joke

kohait00 wrote on Sun, 15 August 2010 09:16
the Get<1>() option (idea from boost?) is a cool trick, but IMHO of little use because it's not a runtime check, but a compile time definition, thus u.a, u.b, u.c is much simpler and clearer in that sense, and less to type anyway. i mean, in terms of compile time specialisation u.Get<1> is same as u.a, you have to provide the index at compiletime, so you know which type.
Yes, it is from boost. As I said, it is mostly useless and the only way it might be helpful is making the code look better and hopefully better readable. But I don't insist on having it at all. At least it learned me some interesting new things about templates

kohait00 wrote on Sun, 15 August 2010 09:16
the Value operator[](int i) is a good idea though. to wrap / unwrap in value (boxing / unboxing is used in C# and others, though there in different context, as base class object).
Don't forget about the GetCount() too I just don't like my implementation of it very much, but I can't come up with anything better. And if possible I would also like to see Begin() and End() implemented, so I could do DUMPC(touple)...

kohait00 wrote on Sun, 15 August 2010 09:16
having Duo, Trio, etc is, as you pointed out, more or less useless, even if it's better to read so i added a 5th T and deaulted past second T. (i'd rater use EmptyClass, but there is no Value(const EmptyClass &) for it, so i used Nuller. might be usefull to have an EmptyClass Value as well?)
I would strongly prefer EmptyClass too. But there might be idealogical problem: Once you make it value compatible, it won't be empty any more Maybe we should do a special class for this purpose, let's say DummyElement, which would be Value and Null compatible.

Apart from what I said above, especially the missing GetCount(), your last code seems reasonable. Definitely not that difficult to read as mine (which is good)

Honza

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 17 Aug 2010 20:22:57 GMT
View Forum Message <> Reply to Message

Quote:
 learned me some interesting new things about templates

i'm always looking to learn as well

Quote:
Don't forget about the GetCount()

i dont think that it is that much of use either. becasue everything is know at compiletime, and Tupel is not meant to be a base class. so i cant imagine any practicle usecase. i'd prefer to, as mirek statet, have it as simple as possible, and serve its due as beeing a compile time known simple strong type container / grouper of data types. not more nor less, except there is more things to have on it, that are of real use.

EmptyClass would be great, but..your point is really good. it wouldnt be empty. but thinking of Nuller as base class in that field is ok too. actually it is a logical requirement, when enabling the Value wrapping of the types. so it's not that tragic.

what do you guys think? is there more to add to this one?
should i add it to bazaar or will it join Others.h?

(I'm trying to 'empty' my development nest, which in time has grown with some little things maybe usefull)

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Tue, 17 Aug 2010 20:52:00 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Tue, 17 August 2010 22:22Quote:Don't forget about the GetCount()i dont think that it is that much of use either. becasue everything is know at compiletime, and Tupel is not meant to be a base class. so i cant imagine any practicle usecase. i'd prefer to, as mirek statet, have it as simple as possible, and serve its due as beeing a compile time known simple strong type container / grouper of data types. not more nor less, except there is more things to have on it, that are of real use.
The main reason I want GetCount() is that it is in every container, from Vector to String. You are

right that everything is known at compile time, but it doesn't really mean that you know everything when you write some code E.g.: Sometimes there are situations where you create templated function which iterates through a container. It doesn't even have to be you who uses the code in the end.

One of the reasons why I like U++ so much is its consistency of interfaces. I feel safe to write template or macro that is doing something general with only basic assumptions about interface, often without knowing what some crazy user of my code will send to it And it is not only about iterating in containers - also ToString (DUMP and LOG are great example of what I tried to describe in previous paragraph), Serialize, Null constructors, operator<<= etc.

Honza

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 17 Aug 2010 21:21:20 GMT
View Forum Message <> Reply to Message

well it's hard to think of Tupel beeing a 'iterable' container.
but i can 'feel' the need somehow as well, especially looking at Value operator[]. so here the type is actually hidden. here is another possib for GetCount()

```
 int GetCount() const
 {
  int c = 2;
  if(typeid(T3) != typeid(Nuller)) ++c;
  if(typeid(T4) != typeid(Nuller)) ++c;
  if(typeid(T5) != typeid(Nuller)) ++c;
  return c;
 }
```

but doesnt save us from things like

```
Typel<int, int, Nuller, float> t; //float type unreachable
for(int i = 0; i<t.GetCount(); i++)
   Value v = t[i];
```

but here, the user is dumb

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Tue, 17 Aug 2010 21:53:52 GMT
View Forum Message <> Reply to Message

I would useint GetCount() const
 {
  if(typeid(T5) != typeid(Nuller)) return 5;
  if(typeid(T4) != typeid(Nuller)) return 4;
  if(typeid(T3) != typeid(Nuller)) return 3;
  return 2;
 }
That works even for the dumb user
Honza

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Thu, 19 Aug 2010 06:30:11 GMT
View Forum Message <> Reply to Message

thats way better. one would need to check for intermediate Nuller though anyway, beeing that the case, why not simply do this

inline int GetCount() const { return 5; }

so it's up to mirek to decide what happens with this stuff

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Mon, 30 Aug 2010 18:57:22 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Thu, 19 August 2010 02:30thats way better. one would need to check for intermediate Nuller though anyway, beeing that the case, why not simply do this

inline int GetCount() const { return 5; }

so it's up to mirek to decide what happens with this stuff

I believe it is not very helpful to mix Tuples with Value or try to pretend they are maps. In fact, ValueArray covers such usage pretty well IMO.

I believe that the common usage scenario is in cases where you need "quick" struct which is not worth defining, like

static Tuple<int, const char *> mapping[] = {
  0, "foo",

```
   1, "bar"
}
```

In fact, the only think to resolve is how to name its members.

Right now, "a, b, c, d, e..." sound like the best option. STL templated indexing seems overengineered to me, "first, second, third..." are too long and "v0, v1, v2, v3..." have "zero index issue".

Well, maybe something like "key, value, value1, value2" would reflect the most typical use.

More ideas? Or letters are it?

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Mon, 30 Aug 2010 20:23:28 GMT
View Forum Message <> Reply to Message

i'd prefer a,b,c,d,e...
its just sweeter and one can give it the meaning it needs, key, value...is too specific already

---

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Mon, 30 Aug 2010 21:13:21 GMT
View Forum Message <> Reply to Message

I agree with the a,b,c,d,e... too. It is nice and simple and fast to write. One more question is how many values should be supported. I personally think that a-f should be about enough...

Honza

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 31 Aug 2010 06:31:31 GMT
View Forum Message <> Reply to Message

if more is needed, this should be possible..

Tupel<int,float,int,float,Tupel<String,char,String,char,double> >

in terms of Tupel as container: Tupel is actually no container..here mirek is right.its a grouper class. anyway, in case such a behaviour is needed, it can be subsequently added deriving.

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 31 Aug 2010 06:44:40 GMT
View Forum Message <> Reply to Message

so here is a cleaned up proposal..

```
template<class T1, class T2, class T3=EmptyClass, class T4=EmptyClass, class
T5=EmptyClass>
class Tupel {
public:
 Tupel(const T1 & a, const T2 & b, const T3 & c, const T4 & d, const T5 & e)
  : a(a), b(b), c(c), d(d), e(e) {}
 Tupel() {}
 T1 a; T2 b; T3 c; T4 d; T5 e;
};
```

Subject: Re: NEW: generic Toupel grouper
Posted by cbpporter on Tue, 31 Aug 2010 07:20:07 GMT
View Forum Message <> Reply to Message

Just a little nitpicking: it is called tuple, unless I am not aware of some British/American alternate spelling.

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 31 Aug 2010 07:44:09 GMT
View Forum Message <> Reply to Message

typedef Tupel Tuple;
no, you're absolutely right.

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Tue, 31 Aug 2010 11:29:26 GMT
View Forum Message <> Reply to Message

dolik.rce wrote on Mon, 30 August 2010 17:13I agree with the a,b,c,d,e... too. It is nice and simple and fast to write. One more question is how many values should be supported. I personally think that a-f should be about enough...

Honza

IMO, 4 is more than enough....

Here is what I got after bit of experimenting:

```
template <typename A, typename B>
struct Tuple2 {
 union {
  A a;
  A key;
 };
 union {
  B b;
  B value;
 };

 int  Compare(const Tuple2& x) const       { return CombineCompare(a, x.a)(b, x.b); }
 bool operator==(const Tuple2& x) const    { return Compare(x) == 0; }
 bool operator!=(const Tuple2& x) const    { return Compare(x) != 0; }
 bool operator<=(const Tuple2& x) const    { return Compare(x) <= 0; }
 bool operator>=(const Tuple2& x) const    { return Compare(x) >= 0; }
 bool operator<(const Tuple2& x) const     { return Compare(x) != 0; }
 bool operator>(const Tuple2& x) const     { return Compare(x) != 0; }

 unsigned GetHashValue() const             { return CombineHash(a, b); }
};

template <typename A, typename B>
inline Tuple2<A, B> MakeTuple(const A& a, const B& b)
{
 Tuple2<A, B> r;
 r.a = a;
 r.b = b;
 return r;
}

template <typename A, typename B, typename C>
struct Tuple3 {
 union {
  A a;
  A key;
 };
 union {
  B b;
  B value;
 };
 union {
  C c;
  C value1;
 };
```

```
 int  Compare(const Tuple3& x) const       { return CombineCompare(a, x.a)(b, x.b)(c, x.c); }
 bool operator==(const Tuple3& x) const    { return Compare(x) == 0; }
 bool operator!=(const Tuple3& x) const    { return Compare(x) != 0; }
 bool operator<=(const Tuple3& x) const    { return Compare(x) <= 0; }
 bool operator>=(const Tuple3& x) const    { return Compare(x) >= 0; }
 bool operator<(const Tuple3& x) const     { return Compare(x) != 0; }
 bool operator>(const Tuple3& x) const     { return Compare(x) != 0; }

 unsigned GetHashValue() const             { return CombineHash(a, b, c); }
};

template <typename A, typename B, typename C>
inline Tuple3<A, B, C> MakeTuple(const A& a, const B& b, const C& c)
{
 Tuple3<A, B, C> r;
 r.a = a;
 r.b = b;
 r.c = c;
 return r;
}

template <typename A, typename B, typename C, typename D>
struct Tuple4 {
 union {
  A a;
  A key;
 };
 union {
  B b;
  B value;
 };
 union {
  C c;
  C value1;
 };
 union {
  D d;
  D value2;
 };

 int  Compare(const Tuple4& x) const       { return CombineCompare(a, x.a)(b, x.b)(c, x.c)(d, x.d); }
 bool operator==(const Tuple4& x) const    { return Compare(x) == 0; }
 bool operator!=(const Tuple4& x) const    { return Compare(x) != 0; }
 bool operator<=(const Tuple4& x) const    { return Compare(x) <= 0; }
 bool operator>=(const Tuple4& x) const    { return Compare(x) >= 0; }
 bool operator<(const Tuple4& x) const     { return Compare(x) != 0; }
 bool operator>(const Tuple4& x) const     { return Compare(x) != 0; }
```

```
 unsigned GetHashValue() const           { return CombineHash(a, b, c, d); }
};

template <typename A, typename B, typename C, typename D>
inline Tuple4<A, B, C, D> MakeTuple(const A& a, const B& b, const C& c, const D& d)
{
 Tuple4<A, B, C, D> r;
 r.a = a;
 r.b = b;
 r.c = c;
 r.d = d;
 return r;
}
```

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Tue, 31 Aug 2010 11:34:12 GMT
View Forum Message <> Reply to Message

Upgrade:

```
template <typename A, typename B>
struct Tuple2 {
 union {
  A a;
  A key;
 };
 union {
  B b;
  B value;
 };

 bool operator==(const Tuple2& x) const    { return a == x.a && b == x.b; }
 bool operator!=(const Tuple2& x) const    { return !operator==(x); }

 int  Compare(const Tuple2& x) const       { return CombineCompare(a, x.a)(b, x.b); }
 bool operator<=(const Tuple2& x) const    { return Compare(x) <= 0; }
 bool operator>=(const Tuple2& x) const    { return Compare(x) >= 0; }
 bool operator<(const Tuple2& x) const     { return Compare(x) != 0; }
 bool operator>(const Tuple2& x) const     { return Compare(x) != 0; }

 unsigned GetHashValue() const             { return CombineHash(a, b); }
};

template <typename A, typename B>
```

```
inline Tuple2<A, B> MakeTuple(const A& a, const B& b)
{
 Tuple2<A, B> r;
 r.a = a;
 r.b = b;
 return r;
}

template <typename A, typename B, typename C>
struct Tuple3 {
 union {
  A a;
  A key;
 };
 union {
  B b;
  B value;
 };
 union {
  C c;
  C value1;
 };

 bool operator==(const Tuple3& x) const    { return a == x.a && b == x.b && c == x.c; }
 bool operator!=(const Tuple3& x) const    { return !operator==(x); }

 int  Compare(const Tuple3& x) const       { return CombineCompare(a, x.a)(b, x.b)(c, x.c); }
 bool operator<=(const Tuple3& x) const    { return Compare(x) <= 0; }
 bool operator>=(const Tuple3& x) const    { return Compare(x) >= 0; }
 bool operator<(const Tuple3& x) const     { return Compare(x) != 0; }
 bool operator>(const Tuple3& x) const     { return Compare(x) != 0; }

 unsigned GetHashValue() const             { return CombineHash(a, b, c); }
};

template <typename A, typename B, typename C>
inline Tuple3<A, B, C> MakeTuple(const A& a, const B& b, const C& c)
{
 Tuple3<A, B, C> r;
 r.a = a;
 r.b = b;
 r.c = c;
 return r;
}

template <typename A, typename B, typename C, typename D>
struct Tuple4 {
 union {
```

```
  A a;
  A key;
 };
 union {
  B b;
  B value;
 };
 union {
  C c;
  C value1;
 };
 union {
  D d;
  D value2;
 };

 bool operator==(const Tuple4& x) const    { return a == x.a && b == x.b && c == x.c && d == x.d; }
 bool operator!=(const Tuple4& x) const    { return !operator==(x); }

 int  Compare(const Tuple4& x) const       { return CombineCompare(a, x.a)(b, x.b)(c, x.c)(d, x.d); }
 bool operator<=(const Tuple4& x) const    { return Compare(x) <= 0; }
 bool operator>=(const Tuple4& x) const    { return Compare(x) >= 0; }
 bool operator<(const Tuple4& x) const     { return Compare(x) != 0; }
 bool operator>(const Tuple4& x) const     { return Compare(x) != 0; }

 unsigned GetHashValue() const             { return CombineHash(a, b, c, d); }
};

template <typename A, typename B, typename C, typename D>
inline Tuple4<A, B, C, D> MakeTuple(const A& a, const B& b, const C& c, const D& d)
{
 Tuple4<A, B, C, D> r;
 r.a = a;
 r.b = b;
 r.c = c;
 r.d = d;
 return r;
}
```

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Tue, 31 Aug 2010 11:36:22 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Tue, 31 August 2010 02:44so here is a cleaned up proposal..

```
template<class T1, class T2, class T3=EmptyClass, class T4=EmptyClass, class
T5=EmptyClass>
class Tupel {
public:
 Tupel(const T1 & a, const T2 & b, const T3 & c, const T4 & d, const T5 & e)
  : a(a), b(b), c(c), d(d), e(e) {}
 Tupel() {}
 T1 a; T2 b; T3 c; T4 d; T5 e;
};
```

There are two troubles:

- sizeof(EmptyClass) == 1, and all these members make it impossible to be initialized with braced list.

- with constructor, again it cannot be initialized with braced list (which, for me, is very important)

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 31 Aug 2010 11:51:01 GMT
View Forum Message <> Reply to Message

nice work,

btw: doensnt ==, !=, etc imply a quite a lot on class capabilities

EmptyClass: what about a DummyClass

```
class DummyClass
{
public:
DummyClass() {}
DummyClass(const DummyClass&) {}
};
```

it even could derive from EmptyClass to compare maybe

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 31 Aug 2010 12:04:26 GMT
View Forum Message <> Reply to Message

void* is simple but makes things possible

template<class A, class B, class C=void*, class D=void*>

```
class Tupel {
public:
 Tupel(const A& a, const B& b, const C& c = NULL, const D& d = NULL)
  : a(a), b(b), c(c), d(d) {}
 Tupel() {}
 A a; B b; C c; D d;
};
```

EDIT: union is problematic, if T=One<int> i.e.

```
template<class A, class B, class C=void*, class D=void*>
class Tupel {
public:
 Tupel(const A& a, const B& b, const C& c = NULL, const D& d = NULL)
  : a(a), b(b), c(c), d(d) {}
 Tupel() {}
 union { A a, key, v1; };
 union { B b, value, v2; };
 union { C c, v3; };
 union { D d, v4; };
};
```

Quote:

error: member 'Upp::One<int> Tupel<int, Upp::One<int>, Upp::String, int>::<anonymous union>::b' with constructor not allowe
 d in union

TDMGCC

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 31 Aug 2010 12:18:20 GMT
View Forum Message <> Reply to Message

i'd suggest to leave it simple..just as u said.

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Tue, 31 Aug 2010 13:21:52 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Tue, 31 August 2010 07:51nice work,

btw: doensnt ==, !=, etc imply a quite a lot on class capabilities

It does, but it is not a problem, because compiler will not instatiate methods unless requested by using them. (Unless I am terribly mistaken .

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Tue, 31 Aug 2010 13:23:43 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Tue, 31 August 2010 08:04
EDIT: union is problematic, if T=One<int> i.e.

Ops, good point. What a pity, I guess I will have to live without 'key'

Mirek

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 31 Aug 2010 13:31:36 GMT
View Forum Message <> Reply to Message

MSC i.e. is not even complaining about mistyped template implementations, as long as they are not instantiated. GCC is better here, it checks implementation. but of corse code instantiation is another topic..

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Tue, 31 Aug 2010 13:38:31 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Tue, 31 August 2010 09:31MSC i.e. is not even complaining about mistyped template implementations, as long as they are not instantiated. GCC is better here, it checks implementation. but of corse code instantiation is another topic..

Not sure what are you speaking about here.

In any case, I have tested

```
struct Foo {
 int x;
};
```

```
CONSOLE_APP_MAIN
{
 Tuple2<int, Foo> x;
 DDUMP(sizeof(x));
}
```

and no problems with both GCC and MSC.

Of course adding

```
x == x;
```

invokes errors in both compilers.

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 31 Aug 2010 13:44:48 GMT
View Forum Message <> Reply to Message

i was refering to the instantiation of template code itself. MSC does not even syntactically check any template untill it's actually used. gcc does check implementation. both instantiate code only on demand.

so you definitely want to stay with Tupel2, Tupel3... with their explicit implementation instead of generic fourpart Tupel, maybe thats better..

what about then to have more literal names, those Duo, Trio, Quartett? (makes code reading nicer IMO).

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Mon, 06 Sep 2010 09:01:16 GMT
View Forum Message <> Reply to Message

Tuple2 - Tuple4 now in Core.

I have added ToString and also "FindTuple" function as compred to posted version.

Subject: Re: NEW: generic Toupel grouper
Posted by mdelfede on Tue, 12 Oct 2010 17:41:12 GMT
View Forum Message <> Reply to Message

Mhhh... it seems to me that Touple hasn't been defined as Moveable, isn't it ?

Would it be possible to make it so :

template <typename A, typename B>
struct Tuple2 :Moveable<Tuple2<A, B> >{

I think tuples are useful inside Vectors and VectorMaps....

Max

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 12 Oct 2010 20:00:58 GMT
View Forum Message <> Reply to Message

hi max,

didn,t think of that, but youre right, should be moveable, because main use was inside containers..

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Wed, 13 Oct 2010 07:07:44 GMT
View Forum Message <> Reply to Message

I am afraid that this is not as easy to make Touple moveable, because that would imply all elements are moveable too, which is not always the case....

It is a little bit troublesome situation. I have no problem requiring all elements to be moveable, however right now I do not see how to add check without breaking POD capability... (the fine method to add check is in destructor, but that would make Touple non-pod).

Subject: Re: NEW: generic Toupel grouper
Posted by mdelfede on Wed, 13 Oct 2010 07:32:07 GMT
View Forum Message <> Reply to Message

luzr wrote on Wed, 13 October 2010 09:07I am afraid that this is not as easy to make Touple moveable, because that would imply all elements are moveable too, which is not always the case....

It is a little bit troublesome situation. I have no problem requiring all elements to be moveable, however right now I do not see how to add check without breaking POD capability... (the fine method to add check is in destructor, but that would make Touple non-pod).

I was afraid about that
Anyways, I think that most Tuple usage is, as you said, to spare a struct definition, and usually with moveable content.

A nice way could be some way to make Tuple moveable on request, but I don't know if that can be implemented (and MoveableTuple would be horrible, of course)

Max

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Wed, 13 Oct 2010 07:41:10 GMT
View Forum Message <> Reply to Message

isnt there any other way to declare TupleX Moveable explicitly whenever its content is Moveable? somehow with template specialisation? so a user wanting to use it in such way simply needs to declare

NTL_MOVEABLE(Tuple2<Foo, Bar>)

and then can use

Vector<Tuple2<Foo, Bar> >

BTW:
i've noticed that the TupleX(const A& a) : a(a) ctors are not used..is there a reason for that? its quite nice to be able to add things like

Array<Tuple<Foo, Bar> > arr;
arr.Add(new Tuple2(foo, bar));

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Wed, 13 Oct 2010 09:50:23 GMT
View Forum Message <> Reply to Message

found sth to work..

NAMESPACE_UPP

//this would be best, but

```
//compile ERROR, preprocessor does misunderstand ','
#if 1
NTL_MOVEABLE(Tuple2<Point, Size>);
#endif

//compile ERROR anyway
#if 0
NTL_MOVEABLE((Tuple2<Point, Size>));
#endif

//this DOES compile
#if 0
typedef Tuple2<Point, Size> BLAA;
NTL_MOVEABLE(BLAA);
#endif

//BOTH compile (got MSC9 and TDMGCC)
#if 0
inline void AssertMoveable0(Tuple2<Point, Size> *) {}
#elif 0
template<> inline void AssertMoveable<Tuple2<Point, Size> >(Tuple2<Point, Size> *) {};
#endif

END_UPP_NAMESPACE

....
//cpp


 Vector<Tuple2<Point, Size> > vv;

 Tuple2<Point, Size>& tps = vv.Add();
 tps.a.x = 100;
 tps.b.cx = 100;
```

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Wed, 13 Oct 2010 10:24:52 GMT
View Forum Message <> Reply to Message

i think the typedef is an easy way, so we can use NTL_MOVEABLE
here a short idea.. if anyone knows how to specify the salt for the name in a better way, welcome..
with this,

```
//V1
```

```
#if 0
#define TUPLE2_MOVEABLE(T1, T2) \
typedef Tuple2<T1,T2> COMBINE3(TD_,T1,T2); \
NTL_MOVEABLE(COMBINE3(TD_,T1,T2)); \

#endif

//V2
#if 1

#define TUPLE2_MOVEABLE(T1, T2, salt) \
typedef Tuple2<T1,T2> TD_salt; \
NTL_MOVEABLE(TD_salt); \

#endif


//V1
#if 0
TUPLE2_MOVEABLE(Point, Size);
#endif

//V2
#if 1
TUPLE2_MOVEABLE(Point, Size, 123);
#endif

//this enables Vector<Tuple2<Point, Size> > to be used
```

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Wed, 13 Oct 2010 16:47:43 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Wed, 13 October 2010 03:32
A nice way could be some way to make Tuple moveable on request, but I don't know if that can be implemented (and MoveableTuple would be horrible, of course)

MoveableTuple - I do not (yet) know the way how to do that either... (I mean, the correct one).

Well, maybe we should just forget about checking moveability of struct components in this particular case...

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Wed, 13 Oct 2010 16:48:32 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Wed, 13 October 2010 06:24i think the typedef is an easy way, so we can use NTL_MOVEABLE
here a short idea.. if anyone knows how to specify the salt for the name in a better way, welcome..
with this,

```
//V1
#if 0
#define TUPLE2_MOVEABLE(T1, T2) \
typedef Tuple2<T1,T2> COMBINE3(TD_,T1,T2); \
NTL_MOVEABLE(COMBINE3(TD_,T1,T2)); \

#endif

//V2
#if 1

#define TUPLE2_MOVEABLE(T1, T2, salt) \
typedef Tuple2<T1,T2> TD_salt; \
NTL_MOVEABLE(TD_salt); \

#endif


//V1
#if 0
TUPLE2_MOVEABLE(Point, Size);
#endif

//V2
#if 1
TUPLE2_MOVEABLE(Point, Size, 123);
#endif

//this enables Vector<Tuple2<Point, Size> > to be used
```

Too verbose, we need something more sexy...

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Wed, 13 Oct 2010 20:03:18 GMT
View Forum Message <> Reply to Message

in any case, the user needs to take care of that all by himself. it's just to make the process to do so a bit easier.
but the thing that TupleX needs to be Moveable at least in many cases is important, otherwise usage in Containers would make no sense..

any provisions on that? any more sexy ones

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Wed, 13 Oct 2010 20:35:46 GMT
View Forum Message <> Reply to Message

luzr wrote on Wed, 13 October 2010 18:48Too verbose, we need something more sexy...
There is a way... a sexy way What would you say to this:template <typename A, typename B>
struct Tuple2 : PossiblyMoveable<Tuple2<A,B>,A,B>{
    ...
}Is that sexy enough? The idea is that PossiblyMoveable<T,A,B,...> evaluates to Moveable<T> if A,B,... are all moveable and to EmptyClass otherwise.

I can't give you the code right now, but I have it implemented (it needs only some cosmetic changes) and I will post it here tonight or tomorrow. Word of warning in advance: It is heavy "template metaprogramming" (Just found that term on wiki, but it fits)

kohait00 wrote on Wed, 13 October 2010 22:03in any case, the user needs to take care of that all by himself. it's just to make the process to do so a bit easier.
As you could already guess from above, what you say is not true It can be done automatically, by abusing the compiler/language just a bit

Honza

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Thu, 14 Oct 2010 00:14:44 GMT
View Forum Message <> Reply to Message

Hi,

As I promised, here is the code I mentioned in my previous post...

Disclaimer: I am aware that the implementation is verbose, bulky, tricky, possibly deceiving and abuses compiler to do things it was never meant to do. Yet I am very happy to see the result - simple template that is able to determine moveability of it's parameters.

Majority of the ugly code is in Topt.h. Since U++ uses two different way how to mark a type moveable (Moveable<> and NTL_MOVEABLE), the code has two ugly parts, one checking each method and ORing both together. Also I had to add some more code to NTL_MOVEABLE macro to be able to determine the moveability correctly. In the Tuple.h, the only change is the

PossiblyMoveable<> base for each size of Tuple.

PossiblyMoveable might be useful in some other context too. I just can't think of any right now. Also IsMoveable<T> and IsBase<Base,Derived> (can determine if Base is really base class of Derived) could be helpful for someone, but they would need to be little more general for common user to use...

Tell me what you think about this, but don't tell me I'm crazy I'm actually quite realistic and don't expect this to get into U++. I wrote it mainly for fun of discovering new tricks and pushing the limits

Best regards,
Honza

File Attachments

1) Topt.h, downloaded 351 times
2) Tuple.h, downloaded 480 times

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Fri, 15 Oct 2010 11:46:48 GMT
View Forum Message <> Reply to Message

Nice and sexy, but makes Tuple non-POD...

As I said, the problem is not Moveable itself, making it a requirement for Tuple would be non-issue.

The problem is that all possible modifications to TEST that both types are moveable are making Tuple non-POD...

Why we like POD:

```
static Tuple2<int, const char *> data[] = {
 { 1, "Kvuli" },
 { 2, "Tomuhle" },
};
```

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Fri, 15 Oct 2010 13:05:58 GMT
View Forum Message <> Reply to Message

so it seems as if the user really needs to provide the grain of compiler information that all of

TupleX parts are Mouveable..

just need to find a sexy way for that, that is not too verbose..

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Fri, 15 Oct 2010 14:24:21 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 15 October 2010 13:46Why we like POD:

```
 static Tuple2<int, const char *> data[] = {
  { 1, "Kvuli" },
  { 2, "Tomuhle" },
 };
```

Point taken... I'll have to read bit more about how to keep a type POD.

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Fri, 15 Oct 2010 15:21:39 GMT
View Forum Message <> Reply to Message

one of my questions above seems to be forgotten

Quote:

BTW:
i've noticed that the TupleX(const A& a) : a(a) ctors are not used..is there a reason for that? its quite nice to be able to add things like
Array<Tuple<Foo, Bar> > arr;
arr.Add(new Tuple2(foo, bar));

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Fri, 15 Oct 2010 17:50:06 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Fri, 15 October 2010 11:21one of my questions above seems to be forgotten

Quote:

BTW:
i've noticed that the TupleX(const A& a) : a(a) ctors are not used..is there a reason for that? its quite nice to be able to add things like
Array<Tuple<Foo, Bar> > arr;
arr.Add(new Tuple2(foo, bar));

The very same reason (POD).

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Mon, 18 Oct 2010 08:00:03 GMT
View Forum Message <> Reply to Message

ok i get it. this one is then possible

Tuple2<int, const char *> data = { 1, "Kvuli" };

, though its a bit more verbose then

Tuple2<int, const char*>(1,"Kvuli);

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Mon, 18 Oct 2010 09:12:48 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Mon, 18 October 2010 10:00ok i get it. this one is then possible

Tuple2<int, const char *> data = { 1, "Kvuli" };

, though its a bit more verbose then

Tuple2<int, const char*>(1,"Kvuli);

The idea is that with Tuple2<...>(a,b) you can't fill the arrays easily, as Mirek was pointing out.

I have read more about the requirements for type to be POD. If I understand right, then the only thing preventing Tuple from being POD and Moveable same time is the inheritance from Moveable. So that leaves us with two options: Either declare the friend void AssertMoveable0(T*) directly in the Tuple struct or doing it outside with (something like) NTL_MOEABLE(). The first approach is troublesome because it is not easy to do that only for tuples with moveable contents. The second option is useless too, since outside of the template is no chance to do it automaticaly, as the template parameters are not known.

Also there might be one more possibility. Is it possible to mark class moveable in other class? If I am not mistaken the entire trick is to make the compiler instantiate the AssertMoveable0(T*). So I would expect something like this to work too: template<class A,class B>
struct helper__Tuple2<A,B>:PossiblyMoveable<Tuple2<A,B>,A,B>{}The helper class should mark Tuple2 as moveable (using the template metaprograming from my last post), while the Tuple2 itself would stay POD. But for some reason I couldn't get it to work so far Any hints?

Honza

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Mon, 18 Oct 2010 10:12:41 GMT
View Forum Message <> Reply to Message

maybe this is sexy enough
it goes just below the TupleX classes respectively, in the Tuple.h

template<class A, class B>
inline void AssertMoveable0(Tuple2<A,B>*) { AssertMoveable<A>(); AssertMoveable<B>(); }

the function only instantiates when Tuple is used inside a container, which needs Moveable<> of TupleX. thus this functions checks if the components are moveable as well, complaining if not, just as usual.

the user wouldn't to care at all.

it at least compiles well for MSC9 and TDMGCC 440, TDMGCC451 and MINGW 440

Subject: Re: NEW: generic Toupel grouper
Posted by dolik.rce on Mon, 18 Oct 2010 11:25:40 GMT
View Forum Message <> Reply to Message

"kohait00 wrote"maybe this is sexy enough
it goes just below the TupleX classes respectively, in the Tuple.h
template<class A, class B>
inline void AssertMoveable0(Tuple2<A,B>*) { AssertMoveable<A>(); AssertMoveable<B>(); }
Yes, that is elegant and simple. Much better than anything I coud think of... I don't see any trouble with it, wonder if Mirek will

Honza

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Tue, 19 Oct 2010 13:21:57 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Mon, 18 October 2010 06:12maybe this is sexy enough
it goes just below the TupleX classes respectively, in the Tuple.h

```
template<class A, class B>
inline void AssertMoveable0(Tuple2<A,B>*) { AssertMoveable<A>(); AssertMoveable<B>(); }
```

the function only instantiates when Tuple is used inside a container, which needs Moveable<> of TupleX. thus this functions checks if the components are moveable as well, complaining if not, just as usual.

the user wouldn't to care at all.

it at least compiles well for MSC9 and TDMGCC 440, TDMGCC451 and MINGW 440

EXCELLENT idea!

Applied, I guess we can now consider Tuples complete. Thanks!

Mirek

Subject: Re: NEW: generic Toupel grouper
Posted by kohait00 on Tue, 07 Dec 2010 14:59:31 GMT
View Forum Message <> Reply to Message

this should speed things up a bit, isn't it?

Tuple.h:30, 65, 102

```
 Tuple3<A, B, C> r = { a, b, c };
```

instead of

```
 Tuple3<A, B, C> r;
r.a = a;
r.b = b;
r.c = c;
```

Subject: Re: NEW: generic Toupel grouper
Posted by mirek on Tue, 07 Dec 2010 20:44:09 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Tue, 07 December 2010 09:59this should speed things up a bit, isn't it?

Tuple.h:30, 65, 102

```
 Tuple3<A, B, C> r = { a, b, c };
```

instead of

```
 Tuple3<A, B, C> r;
r.a = a;
r.b = b;
r.c = c;
```

Sorry, but nonsense. No speedup, and there are (AFAIK) some pretty restrictive rules about {} initialization in C++...