Subject: ConvertInt > templatable Convert<T>
Posted by kohait00 on Fri, 07 Aug 2009 14:17:57 GMT
View Forum Message <> Reply to Message

hi there

I came along to need a Min Max EditValue control, but specificable as template, so to speak.

instead of

EditInt edit, or EditDouble edit

i needed

MyValueEdit<T> edit;

In code speaking, it should be like

```
template<class T>
class MyEditMinMax : public EditMinMax<T, Convert<T>>
{};
```

instead of

```
typedef EditMinMax<T, ConvertInt> EditInt
```

so i needed kind of a Convert<T>
to make this possible.

looking at the code, i found out that the Convert Datatype is used as Base some base levels deeper, but ther then as typesafe.

it would be great to have the Convert<T> implementation. i know it would take much recoding, and the api would change, also the speed penalty for stuff like

```
if(typeid(T)) == typeid(int))
invoke some int specific convert function
```

is there any intelligent way to do that?

known issues to that:
1. Convert uses a typesafe parametrized standard constructor in ConvertInt (cause of some

constants maybe) and the like. this might need to be split, a standard constructor invoking an init function or so..

2. the converters call each another scan founction and implement their own format functions, this might need to form differently.

any comments?

Just keep in mind, what is needed is just a EditMinMax<T> or so, instead of EditMinMax<T, ConvertInt>
the class looks for a fitting converter itself, invoking its functions.

meanwhile, i try to solve it by now, with havnig stuff like that:
there, maybe you'll see and understand better the problem

```
/// an own Edit Control with min max, using Converters in a different way

template <class DataType/*, class Cv*/>
class MyEditValue : public EditField /*, public Cv*/{

protected:
 //as long as we do not have a Convert<T> need to mirror that
 Convert * pcv;

public:
 MyEditValue& operator=(const DataType& t)  { SetData(t); return *this; }
 operator DataType() const                  { return GetData(); }

 MyEditValue()
 {
  pcv = NULL;
  if(  (typeid(DataType) == typeid(int))
    || (typeid(DataType) == typeid(unsigned int))
    || (typeid(DataType) == typeid(short))
    || (typeid(DataType) == typeid(unsigned short))
    || (typeid(DataType) == typeid(char))
    || (typeid(DataType) == typeid(unsigned char))
   )
  {
   pcv = new ConvertInt();
  }
  else if( (typeid(DataType) == typeid(double))
    || (typeid(DataType) == typeid(float))
   )
  {
   pcv = new ConvertDouble();
  }
```

```
 else if( (typeid(DataType) == typeid(long))
   || (typeid(DataType) == typeid(unsigned long))
  )
 {
  pcv = new ConvertInt64();
 }
 else
 {
  D6("MyEditValue","unknown type : " << typeid(DataType).name());
  D6("MyEditValue","not covertible: " << s);
  return;
 }

 if(pcv != NULL)
  SetConvert(*pcv); //instead *this
}

virtual ~MyEditValue()
{
 if(pcv != NULL)
  delete pcv;
 pcv = NULL;
}

DataType GetMax()
{
 if (!pcv) return (DataType)0;

 if(  (typeid(DataType) == typeid(int))
   || (typeid(DataType) == typeid(unsigned int))
   || (typeid(DataType) == typeid(short))
   || (typeid(DataType) == typeid(unsigned short))
   || (typeid(DataType) == typeid(char))
   || (typeid(DataType) == typeid(unsigned char))
  )
 {
  return ((ConvertInt*)pcv)->GetMax();
 }
 else if( (typeid(DataType) == typeid(double))
   || (typeid(DataType) == typeid(float))
  )
 {
  return ((ConvertDouble*)pcv)->GetMax();
 }
 else if( (typeid(DataType) == typeid(long))
   || (typeid(DataType) == typeid(unsigned long))
  )
 {
```

```
  return ((ConvertInt64*)pcv)->GetMax();
 }
 else
 {
  D6("MyEditMinMax","unknown type : " << typeid(DataType).name());
  D6("MyEditMinMax","not covertible: " << s);
  return (DataType)0;
 }
}

DataType GetMin()
{
 if (!pcv) return (DataType)0;

 if(  (typeid(DataType) == typeid(int))
   || (typeid(DataType) == typeid(unsigned int))
   || (typeid(DataType) == typeid(short))
   || (typeid(DataType) == typeid(unsigned short))
   || (typeid(DataType) == typeid(char))
   || (typeid(DataType) == typeid(unsigned char))
  )
 {
  return ((ConvertInt*)pcv)->GetMin();
 }
 else if( (typeid(DataType) == typeid(double))
   || (typeid(DataType) == typeid(float))
  )
 {
  return ((ConvertDouble*)pcv)->GetMin();
 }
 else if( (typeid(DataType) == typeid(long))
   || (typeid(DataType) == typeid(unsigned long))
  )
 {
  return ((ConvertInt64*)pcv)->GetMin();
 }
 else
 {
  D6("MyEditMinMax","unknown type : " << typeid(DataType).name());
  D6("MyEditMinMax","not covertible: " << s);
  return (DataType)0;
 }
}

};

template <class DataType/*, class Cv*/>
class MyEditMinMax : public MyEditValue<DataType/*, Cv*/> {
```

```
public:
 MyEditMinMax& operator=(const DataType& t)          { SetData(t); return *this; }

 MyEditMinMax() {}
 MyEditMinMax(DataType min, DataType max)
 {
  //for the converts a *typesafe* call is needed.
  //as long as we dont have Converter<T>::MinMax(), we need to mirror that
  //Cv::MinMax(min, max);

  if(  (typeid(DataType) == typeid(int))
    || (typeid(DataType) == typeid(unsigned int))
    || (typeid(DataType) == typeid(short))
    || (typeid(DataType) == typeid(unsigned short))
    || (typeid(DataType) == typeid(char))
    || (typeid(DataType) == typeid(unsigned char))
   )
  {
   ((ConvertInt*)MyEditValue<DataType>::pcv)->MinMax(min, max);
  }
  else if( (typeid(DataType) == typeid(double))
    || (typeid(DataType) == typeid(float))
   )
  {
   ((ConvertDouble*)MyEditValue<DataType>::pcv)->MinMax(min, max);
  }
  else if( (typeid(DataType) == typeid(long))
    || (typeid(DataType) == typeid(unsigned long))
   )
  {
   ((ConvertInt64*)MyEditValue<DataType>::pcv)->MinMax(min, max);
  }
  else
  {
   D6("MyEditMinMax","unknown type : " << typeid(DataType).name());
   D6("MyEditMinMax","not covertible: " << s);
  }

 }

 MyEditMinMax& Min(DataType min)
 {
  //Cv::Min(min); return *this;
  if(  (typeid(DataType) == typeid(int))
    || (typeid(DataType) == typeid(unsigned int))
    || (typeid(DataType) == typeid(short))
    || (typeid(DataType) == typeid(unsigned short))
    || (typeid(DataType) == typeid(char))
```

```
   || (typeid(DataType) == typeid(unsigned char))
  )
 {
  ((ConvertInt*)MyEditValue<DataType>::pcv)->Min(min);
 }
 else if( (typeid(DataType) == typeid(double))
   || (typeid(DataType) == typeid(float))
  )
 {
  ((ConvertDouble*)MyEditValue<DataType>::pcv)->Min(min);
 }
 else if( (typeid(DataType) == typeid(long))
   || (typeid(DataType) == typeid(unsigned long))
  )
 {
  ((ConvertInt64*)MyEditValue<DataType>::pcv)->Min(min);
 }
 else
 {
  D6("MyEditMinMax","unknown type : " << typeid(DataType).name());
  D6("MyEditMinMax","not covertible: " << s);
 }
 return *this;
}

MyEditMinMax& Max(DataType max)
{
 //Cv::Max(max); return *this;
 if(  (typeid(DataType) == typeid(int))
   || (typeid(DataType) == typeid(unsigned int))
   || (typeid(DataType) == typeid(short))
   || (typeid(DataType) == typeid(unsigned short))
   || (typeid(DataType) == typeid(char))
   || (typeid(DataType) == typeid(unsigned char))
  )
 {
  ((ConvertInt*)MyEditValue<DataType>::pcv)->Max(max);
 }
 else if( (typeid(DataType) == typeid(double))
   || (typeid(DataType) == typeid(float))
  )
 {
  ((ConvertDouble*)MyEditValue<DataType>::pcv)->Max(max);
 }
 else if( (typeid(DataType) == typeid(long))
   || (typeid(DataType) == typeid(unsigned long))
  )
 {
```

```
   ((ConvertInt64*)MyEditValue<DataType>::pcv)->Max(max);
  }
  else
  {
   D6("MyEditMinMax","unknown type : " << typeid(DataType).name());
   D6("MyEditMinMax","not covertible: " << s);
  }
  return *this;
 }

 MyEditMinMax& NotNull(bool nn = true)
 {
  //Cv::NotNull(nn); return *this;
  if(  (typeid(DataType) == typeid(int))
    || (typeid(DataType) == typeid(unsigned int))
    || (typeid(DataType) == typeid(short))
    || (typeid(DataType) == typeid(unsigned short))
    || (typeid(DataType) == typeid(char))
    || (typeid(DataType) == typeid(unsigned char))
   )
  {
   ((ConvertInt*)MyEditValue<DataType>::pcv)->NotNull(nn);
  }
  else if( (typeid(DataType) == typeid(double))
    || (typeid(DataType) == typeid(float))
   )
  {
   ((ConvertDouble*)MyEditValue<DataType>::pcv)->NotNull(nn);
  }
  else if( (typeid(DataType) == typeid(long))
    || (typeid(DataType) == typeid(unsigned long))
   )
  {
   ((ConvertInt64*)MyEditValue<DataType>::pcv)->NotNull(nn);
  }
  else
  {
   D6("MyEditMinMax","unknown type : " << typeid(DataType).name());
   D6("MyEditMinMax","not covertible: " << s);
   return;
  }
 }
};


///
```

Subject: Re: ConvertInt > templatable Convert<T>
Posted by mrjt on Fri, 07 Aug 2009 16:03:08 GMT
View Forum Message <> Reply to Message

I'm not really sure what problem you're trying to solve with this, though I think I see what you're trying to do.

I think there are simpler ways of achieving it though, for instance:

```
template <class DataType>
class ConvertT : public Convert
{
private:
 DataType minval;
 DataType maxval;
 bool notnull;

public:
 ConvertT() : minval(Null), maxval(Null), notnull(false) {}

 virtual Value Scan(const Value& text) const {
  if (IsNotNull() && IsNull(text)) return NotNullError();
  Value v;
  if(  (typeid(DataType) == typeid(int))
    || (typeid(DataType) == typeid(unsigned int))
    || (typeid(DataType) == typeid(short))
    || (typeid(DataType) == typeid(unsigned short))
    || (typeid(DataType) == typeid(char))
    || (typeid(DataType) == typeid(unsigned char))
   )
  {
   v = StdConvertInt().Scan(text);
  }
  else if( (typeid(DataType) == typeid(double))
    || (typeid(DataType) == typeid(float))
   )
  {
   v = StdConvertDouble().Scan(text);
  }
  else if( (typeid(DataType) == typeid(uint64))
    || (typeid(DataType) == typeid(int64))
   )
  {
```

```
   v = ConvertInt64().Scan(text); // No StdConvert64
  }
  else
   v = StdConvert().Scan(text);

  if (!IsNull(v) && !IsNull(minval) && !IsNull(maxval)) {
   DataType m = DataType(v);
   if(m >= minval && m <= maxval)
    return v;
   return ErrorValue(UPP::Format(t_("Number must be between %s and %s."), Format(minval),
Format(maxval)));
  }
  return v;
 }
 virtual Value Format(const Value& q) const {
              // You may want to check the type of q here to ensure it matches DataType
  return StdConvert().Format(q);
 }

 ConvertT<DataType>& MinMax(DataType _min, DataType _max) { minval = _min; maxval =
_max; return *this; }
 ConvertT<DataType>& Min(DataType _min)              { minval = _min; return *this; }
 ConvertT<DataType>& Max(DataType _max)               { maxval = _max; return *this; }
 ConvertT<DataType>& NotNull(bool b = true)          { notnull = b; return *this; }
 ConvertT<DataType>& NoNotNull()                     { return NotNull(false); }
 DataType         GetMin() const                 { return minval; }
 DataType         GetMax() const                  { return maxval; }
 bool         IsNotNull() const               { return notnull; }
};
```

And then use templated edit ctrls like so:

```
template <class T>
struct EditMinMaxT : public EditMinMax<T, ConvertT<T> >
{ };
```

Subject: Re: ConvertInt > templatable Convert<T>
Posted by kohait00 on Thu, 20 Aug 2009 08:18:18 GMT
View Forum Message <> Reply to Message

thanks, i'l accomodate the code, maybe we can include it in Upp also

---

Subject: this one worked for me

had to change it slightly..code is found at bottom

i havent been able to test it extensivly, but it seems to work quite well..but so far it works for int and float

so incase anyone wants to have a number EditField without specifying Type directly..

so instead of

EditInt edit;

you can use

EditMinMax<DataType, ConvertT<DataType> > edit;

or event better

template <class DataType>
EditNumber : public EditMinMax<DataType, ConvertT<DataType> > {}

EditNumber<float> edit;
EditNumber<T> edit2;

now here is the code that i could compile.
thanks for help

```
template <class DataType>
class ConvertT : public Convert
{
private:
 DataType minval;
 DataType maxval;
 bool notnull;

public:
 //ConvertT() : minval(Null), maxval(Null), notnull(false) {}
 ConvertT()
 {
  if(  (typeid(DataType) == typeid(int))
    || (typeid(DataType) == typeid(short))
    || (typeid(DataType) == typeid(char))
```

```
   || (typeid(DataType) == typeid(int64))
  )
 {
  minval = (DataType) (1<<(8*sizeof(DataType)-1)); //0x8000...
  maxval = (DataType) (minval - 1);//0x7FFF..
  notnull = true;
 }

 else if(  (typeid(DataType) == typeid(unsigned int))
   || (typeid(DataType) == typeid(unsigned short))
   || (typeid(DataType) == typeid(unsigned char))
   || (typeid(DataType) == typeid(uint64))
  )
 {
  minval = 0;
  maxval = (DataType) (-1); //FFFF...
  notnull = true;
 }

 else
 {
  minval = (DataType)0; //Null;
  maxval = (DataType)0; //Null;
  notnull = false;
 }

}

virtual Value Scan(const Value& text) const {
 if (IsNotNull() && IsNull(text)) return NotNullError();
 Value v;
 if(  (typeid(DataType) == typeid(int))
   || (typeid(DataType) == typeid(unsigned int))
   || (typeid(DataType) == typeid(short))
   || (typeid(DataType) == typeid(unsigned short))
   || (typeid(DataType) == typeid(char))
   || (typeid(DataType) == typeid(unsigned char))
  )
 {
  v = StdConvertInt().Scan(text);
 }
 else if( (typeid(DataType) == typeid(double))
   || (typeid(DataType) == typeid(float))
  )
 {
  v = StdConvertDouble().Scan(text);
 }
 else if( (typeid(DataType) == typeid(uint64))
```

```
    || (typeid(DataType) == typeid(int64))
   )
  {
   v = ConvertInt64().Scan(text); // No StdConvert64
  }
  else
   v = StdConvert().Scan(text);

  if ( !IsNull(v)
   //&&  !IsNull(minval)
   //&&  !IsNull(maxval)
   && IsNotNull()
   )
  {
   DataType m;

   if(  (typeid(DataType) == typeid(int))
     || (typeid(DataType) == typeid(unsigned int))
     || (typeid(DataType) == typeid(short))
     || (typeid(DataType) == typeid(unsigned short))
     || (typeid(DataType) == typeid(char))
     || (typeid(DataType) == typeid(unsigned char))
    )
   {
    m = (DataType)(int)(v);
   }
   else if( (typeid(DataType) == typeid(double))
     || (typeid(DataType) == typeid(float))
    )
   {
    m = (DataType)(double)(v);
   }
   else if( (typeid(DataType) == typeid(uint64))
     || (typeid(DataType) == typeid(int64))
    )
   {
    m = (DataType)(int64)(v);
   }
   else
   {
    return ErrorValue(UPP::Format(t_("Number must be a valid type. %s"),
Format((DataType)(0)))); //gives compile errors if no second format
   }


   if(m >= minval && m <= maxval)
    return v;
   return ErrorValue(UPP::Format(t_("Number must be between %s and %s."), Format(minval),
```

```
Format(maxval)));
  }
  return v;
 }
 virtual Value Format(const Value& q) const
 {
  // You may want to check the type of q here to ensure it matches DataType
  return StdConvert().Format(q);
 }

 ConvertT<DataType>& MinMax(DataType _min, DataType _max) { minval = _min; maxval =
_max; return *this; }
 ConvertT<DataType>& Min(DataType _min)              { minval = _min; return *this; }
 ConvertT<DataType>& Max(DataType _max)              { maxval = _max; return *this; }
 ConvertT<DataType>& NotNull(bool b = true)          { notnull = b; return *this; }
 ConvertT<DataType>& NoNotNull()                     { return NotNull(false); }
 DataType         GetMin() const                 { return minval; }
 DataType         GetMax() const                 { return maxval; }
 bool         IsNotNull() const              { return notnull; }
};
```

Subject: Re: this one worked for me
Posted by mrjt on Thu, 20 Aug 2009 14:26:00 GMT
View Forum Message <> Reply to Message

Well, the problem is that you are trying to use types that cannot have a Null defined for them (this is usually the case for smaller variants of a type, float and short for instance). These cannot be directly converted to and from Values.

I actually should have removed all of these types from my version as it won't work compile (as you discovered). Hacking in arbitrary Null values isn't a very good solution IMO.

What I hadn't thought of before was that the task would be much more easily accomplished with something like:struct ConvertUnsignedChar : public ConvertInt

```
{
 ConvertUnsignedChar() { MinMax(0, 255); }
};

template <class CONVERT, const int CHARS>
struct EditCustom : public EditMinMax<int, CONVERT >
{
 EditCustom() { MaxChars(CHARS); }
};
```

typedef EditCustom<ConvertUnsignedChar, 3> EditUnsignedChar;(This would be even simpler if you didn't need the max chars)

Since you're doing all the RTTI typeid stuff internally anyway, why not just do it in a function somewhere to create the correct type of EditField/Convert and avoid all the horrific bodging?