Subject: Anonymous delegates
Posted by Factor on Mon, 22 Oct 2007 14:21:13 GMT
View Forum Message <> Reply to Message

I've written a simple header file with some macros and classes to implement a primitive form of anonymous delegates. It also contains some sort of foreach macro to use with UPP container classes.
I'll extend it in the future. Maybe somebody finds it usefull.

Example:

```
DELEGATE(button, WhenAction,
  _this.Title("Title changed!");
);

DELEGATE1(button, WhenAction,
  String, s, "Parameter test",
  {
     PromptOK(s);
     _this.Title(s);
  });
...
Vector<String> list;
...
foreach(String v,list, PromptOK(v));

String s;
foreach(String v, list,
  s+=v;
  PromptOK(s);
);
```

File Attachments

1) delegates.h, downloaded 447 times

Subject: Re: Anonymous delegates
Posted by mirek on Tue, 30 Oct 2007 00:26:01 GMT
View Forum Message <> Reply to Message

Hm, interesting. However, I guess that the problem here is that delegate code cannot simply access its "owner" elements (attributes, methods).

One possible solution would be to pass 'this' as some implicit parameter, but that is still far from

perfect because of access control (private:, protected:).

Mirek

Subject: Re: Anonymous delegates
Posted by Zardos on Wed, 07 Nov 2007 22:21:41 GMT
View Forum Message <> Reply to Message

Hi,

I have seen your foreach macro. Because I think it does not blend very good into the c++ syntax I would like to share my UPP-foreach version:

```
#define loop(v) \
 int MK__s = v; for(int _lv_ = MK__s; _lv_ > 0; _lv_--)

#define loopi(n, v) \
 int MK__s = v; for(int n = 0; n < MK__s; n++)

#define foreach(e, arr) \
 int MK__s = (arr).GetCount(); for(int _lv_ = 0; _lv_ < MK__s; _lv_++) \
 if(bool _foreach_continue = true) \
 for(e = (arr)[_lv_]; _foreach_continue; _foreach_continue = false)

#define foreach_n(n, e, arr) \
 int MK__s = (arr).GetCount(); for(int _lv_ = (n); _lv_ < MK__s; _lv_++) \
 if(bool _foreach_continue = true) \
 for(e = (arr)[_lv_]; _foreach_continue; _foreach_continue = false)

#define foreach_rev(e, arr) \
 for(int _lv_ = (arr).GetCount() - 1; _lv_ >= 0; _lv_--) \
 if(bool _foreach_continue = true) \
 for(e = (arr)[_lv_]; _foreach_continue; _foreach_continue = false)
```

Examples:

```
// repeat N times:
loop(10) {
   printf("Hello World\n");
}

repeat N times with index:
loopi(i, 10) {
   printf("Hello World %d\n", i);
}
```

```
// access container elements
Vector<int> vec;
foreach(int e, vec) {
   printf("e = %d\n", e);
}

// by ref:
Vector<int> vec;
foreach(int &e, vec) {
   printf("e = %d\n", e);
}

// in reverse order:
Vector<String> vec;
foreach_rev(const String &e, vec) {
   printf("e = %s\n", e);
}

// e declared outside:

Vector<int> vec;
int e;
foreach(e, vec) {
   printf("e = %d\n", e);
}

// e is only visible inside the foreach_scope:
Vector<String> vec;
foreach(const String &e, vec)
   printf("e = %s\n", e);

foreach_rev(const String &e, vec) // e used again
   printf("e = %s\n", e);
```

The macro produces "optimal" code if compiled in Release mode (VC++ / MINGW). Produces larger code in debug mode than a handcoded loop.

- Ralf

Subject: Re: Anonymous delegates
Posted by unodgs on Wed, 07 Nov 2007 22:29:19 GMT
View Forum Message <> Reply to Message

Hi Zardos. Thanks for sharing it. I'll start to use it!

Subject: Re: Anonymous delegates
Posted by Zardos on Thu, 08 Nov 2007 02:17:54 GMT
View Forum Message <> Reply to Message

If you use it, please be aware it's still only a macro.

To illustrate the fundamentel problem a simple example:

```
Vector<int> CreateResultVector() {
  Vector<int> r;
  r.Add(1);
  r.Add(2);
  r.Add(3);
  return r;
}

...


foreach(int e, CreateResultVector())
  DUMP(e);
```

Basically the code is stupidly translated to something like this:

```
for(int i = 0; i < CreateResultVector().GetCount(); i++)
  DUMP(CreateResultVector()[i]);
```

... CreateResultVector is called multiple times!
But this is probably not what you would expect from a real foreach build into the language!

If you want to avoid this I recommend the following macros and templates (it's becoming ugly, now...):

```
// a simple type wrapper
template<class T> struct Type2Type {};

// convert an expression of type T to an expression of type Type2Type<T>
template<class T>
Type2Type<T> EncodeType(T const & t) {
 return Type2Type<T>();
}

// convertible to Type2Type<T> for any T
struct AnyType {
 template<class T>
 operator Type2Type<T>() const { return Type2Type<T>(); }
```

```
};

struct IterHolder {
 void *p;
 void *x;

 template<class T> Begin(const T& v)               { p = (void*)v.Begin(); x = (void*)v.End(); }
 template<class T> End(const T& v)                 { p = (void*)((v.End()) - 1); x = (void*)v.Begin(); }
 template<class T> Prev(Type2Type<T>)                  { p = ((T*)p) - 1; }
 template<class T> Next(Type2Type<T>)                  { p = ((T*)p) + 1; }
                   bool CheckF() const             { return p < x; }
                   bool CheckB() const             { return p >= x; }
 template<class T> T& Get(Type2Type<T>) const        { return *((T*)p); }
};

// convert an expression of type T to an expression of type Type2Type<T> without evaluating the
expression
#define ENCODED_TYPEOF( container ) \
  ( true ? AnyType() : EncodeType( container ) )


#define loop(v) \
 int MK__s = v; for(int _lv_ = MK__s; _lv_ > 0; _lv_--)

#define loopi(n, v) \
 int MK__s = v; for(int n = 0; n < MK__s; n++)

#define foreach(e, arr) \
 IterHolder MK__s; for(MK__s.Begin(arr); MK__s.CheckF();
MK__s.Next(ENCODED_TYPEOF(arr[0]))) \
 if(bool _foreach_continue = true) \
 for(e = MK__s.Get(ENCODED_TYPEOF(arr[0])); _foreach_continue; _foreach_continue = false)

#define foreach_rev(e, arr) \
 IterHolder MK__s; for(MK__s.End(arr); MK__s.CheckB();
MK__s.Prev(ENCODED_TYPEOF(arr[0]))) \
 if(bool _foreach_continue = true) \
 for(e = MK__s.Get(ENCODED_TYPEOF(arr[0])); _foreach_continue; _foreach_continue = false)
```

This version evaluates CreateResultVector only once...

Surprisingly this version is even faster than the previous version I posted (in release mode).

For example the following code:

```
 UTest(foreach) {
  Vector<int> vec;
  vec.Insert(0, 1, 10000000);
```

```
 int c = 0;
 loop(10) {
  foreach(int qq, vec) {
   c += qq;
  }
 }
 UCheck(c == 100000000);
}
```

is as fast as:

```
UTest(Iteration) {
 Vector<int> vec;
 vec.Insert(0, 1, 10000000);
 int c = 0;
 loop(10) {
  const int *e = vec.End();
  for(const int *it = vec.Begin(); it < e; it++) {
   c += *it;
  }
 }
 UCheck(c == 100000000);
}
```

BTW the basic idea is from: http://www.artima.com/cppsource/foreach.html

- Ralf

Subject: Re: Anonymous delegates
Posted by mirek on Wed, 05 Dec 2007 14:33:20 GMT
View Forum Message <> Reply to Message

Zardos wrote on Wed, 07 November 2007 21:17If you use it, please be aware it's still only a macro.

To illustrate the fundamentel problem a simple example:

```
Vector<int> CreateResultVector() {
  Vector<int> r;
  r.Add(1);
  r.Add(2);
  r.Add(3);
  return r;
```

```
}

...


foreach(int e, CreateResultVector())
  DUMP(e);


Basically the code is stupidly translated to something like this:

for(int i = 0; i < CreateResultVector().GetCount(); i++)
  DUMP(CreateResultVector()[i]);


... CreateResultVector is called multiple times!
But this is probably not what you would expect from a real foreach build into the language!


Curiously, this is exactly what I would expect... (I mean, called multiple times).

Mirek
```

Subject: Re: Anonymous delegates
Posted by Zardos on Wed, 05 Dec 2007 20:33:43 GMT
View Forum Message <> Reply to Message

luzr wrote on Wed, 05 December 2007 15:33Zardos wrote on Wed, 07 November 2007 21:17If you use it, please be aware it's still only a macro.

To illustrate the fundamentel problem a simple example:

```
Vector<int> CreateResultVector() {
  Vector<int> r;
  r.Add(1);
  r.Add(2);
  r.Add(3);
  return r;
}

...


foreach(int e, CreateResultVector())
  DUMP(e);
```

Basically the code is stupidly translated to something like this:

```
for(int i = 0; i < CreateResultVector().GetCount(); i++)
   DUMP(CreateResultVector()[i]);
```

... CreateResultVector is called multiple times!
But this is probably not what you would expect from a real foreach build into the language!

Curiously, this is exactly what I would expect... (I mean, called multiple times).

Mirek

Yes, but you are a very experienced c++ programmer. You know "foreach" is a macro and simply expect "macro behaviour".

If a "foreach" would be available in c++ it would probably evaluate CreateResultVector only once like in C#, Python or Ruby.

For example the following ruby code evaluates create_result_vector only once:

```
def create_result_vector
 [1, 2, 3]
end

for e in create_result_vector do
 puts e
end
```

- Ralf

Subject: Re: Anonymous delegates
Posted by mirek on Fri, 07 Dec 2007 08:36:03 GMT
View Forum Message <> Reply to Message

```
// a simple type wrapper
template<class T> struct Type2Type {};

// convert an expression of type T to an expression of type Type2Type<T>
template<class T>
Type2Type<T> EncodeType(T const & t) {
  return Type2Type<T>();
}
```

```
// convertible to Type2Type<T> for any T
struct AnyType {
 template<class T>
 operator Type2Type<T>() const { return Type2Type<T>(); }
};

struct IterHolder {
 void *p;
 void *x;

 template<class T> Begin(const T& v)               { p = (void*)v.Begin(); x = (void*)v.End(); }
 template<class T> End(const T& v)                 { p = (void*)((v.End()) - 1); x = (void*)v.Begin(); }
 template<class T> Prev(Type2Type<T>)                  { p = ((T*)p) - 1; }
 template<class T> Next(Type2Type<T>)                  { p = ((T*)p) + 1; }
                bool CheckF() const              { return p < x; }
                bool CheckB() const              { return p >= x; }
 template<class T> T& Get(Type2Type<T>) const        { return *((T*)p); }
};

// convert an expression of type T to an expression of type Type2Type<T> without evaluating the
expression
#define ENCODED_TYPEOF( container ) \
  ( true ? AnyType() : EncodeType( container ) )


#define loop(v) \
 int MK__s = v; for(int _lv_ = MK__s; _lv_ > 0; _lv_--)

#define loopi(n, v) \
 int MK__s = v; for(int n = 0; n < MK__s; n++)

#define foreach(e, arr) \
 IterHolder MK__s; for(MK__s.Begin(arr); MK__s.CheckF();
MK__s.Next(ENCODED_TYPEOF(arr[0]))) \
 if(bool _foreach_continue = true) \
 for(e = MK__s.Get(ENCODED_TYPEOF(arr[0])); _foreach_continue; _foreach_continue = false)

#define foreach_rev(e, arr) \
 IterHolder MK__s; for(MK__s.End(arr); MK__s.CheckB();
MK__s.Prev(ENCODED_TYPEOF(arr[0]))) \
 if(bool _foreach_continue = true) \
 for(e = MK__s.Get(ENCODED_TYPEOF(arr[0])); _foreach_continue; _foreach_continue = false)
```

Well, is not C++ fun?

Anyway, IMO this "foreach" has problem:

```
if(x)
  foreach(int a, v)
```

Mirek

Subject: Re: Anonymous delegates
Posted by Zardos on Fri, 07 Dec 2007 10:07:23 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 07 December 2007 09:36
Well, is not C++ fun?
Mirek

I'm a split personality about my C++ opinion.
On one day I think C++ is one of the most horrible languages ever invented.
And on another day I'm amazed and impressed about how much thought has been put into the language...

luzr wrote on Fri, 07 December 2007 09:36
Anyway, IMO this "foreach" has problem:

```
if(x)
  foreach(int a, v)
```

Mirek

Yes you are right! Thats not nice!
I currently can not test the code, but I think this should solve the problem:

```
struct IterHolder {
 void *p;
 void *x;

 IterHolder(bool) {}
 operator bool() const                          { return false; }

 template<class T> Begin(const T& v)              { p = (void*)v.Begin(); x = (void*)v.End(); }
 template<class T> End(const T& v)               { p = (void*)((v.End()) - 1); x = (void*)v.Begin(); }
 template<class T> Prev(Type2Type<T>)               { p = ((T*)p) - 1; }
 template<class T> Next(Type2Type<T>)               { p = ((T*)p) + 1; }
                bool CheckF() const             { return p < x; }
                bool CheckB() const             { return p >= x; }
 template<class T> T& Get(Type2Type<T>) const       { return *((T*)p); }
```

```
};

#define foreach(e, arr) \
 if(IterHolder _ith_ = false) {} else \
  for(_ith_.Begin(arr); _ith_.CheckF(); _ith_.Next(ENCODED_TYPEOF(arr[0]))) \
   if(bool _foreach_continue = true) \
    for(e = _ith_.Get(ENCODED_TYPEOF(arr[0])); _foreach_continue; _foreach_continue = false)
```

I'm not sure if the c++ optimizer can still remove all the noise and create a simple iterater loop for this version. But I guess performance should still be the same as a hand written loop.

Before using this code in production I probably would tweak the IterHolder and the Upp containers a little bit and make it more generic. I already have written 4 version of foreach and currently using a slightly different version, but I get tired of it... The concept is always the same. The main trick is ENCODED_TYPEOF(...) to get a the type of an expression without evaluating it.

- Ralf
