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