Lambda Terms

C^{++} -11 has lambda expressions:

```
auto square = [] ( double x ) { return x * x; };
std::cout << square( 2.0 ) << "\n";
std::cout << square( 3.0 ) << "\n";
```

Lambda expressions are not as special as you think.

They are syntactic variation of something that you know already:

```
class T1234
{
   double operator( ) const ( double x ) { return x * x; }
};
```

So we have

```
   T1234 square;

   std::cout << square( 2.0 ) << "\n";
   std::cout << square( 3.0 ) << "\n";
```

Type of Lambda

Every lambda expression has its unique type:

```
auto square = [] ( double d ) { return d * d; };
   // T1234
auto square2 = [] ( double d ) { return d * d; };
   // T1235
auto square3 = [] ( double d ) { return d * d; };
   // T1236

square = square2; // Won't go.
```

Using `std::function< >` (2)

If you want a type that you can assign to, use `std::function< F( A_1, ..., An ) >`, where `F` is the return type, and `A1, ..., An` are the argument types.

You have to `#include <functional>`

```
using T = std::function< double( double ) > ;
   // Use T as abbreviation.

T square = [] ( double d ) { return d * d; };
T square2 [] ( double d ) { return d * d; };

square = square2;
square = [] ( double d ) { return d * d; };
```

Replacing the lambdas by explicit class definition will also work:

```
struct sq
{
   double operator( ) ( double d ) const
   {
      return d*d;
   }
};


square = sq( );         // Will compile.
```

`std::function< F( A_1, ..., An ) >` is a kind of smart pointer.

It can be **nullptr**.

Plotting a Function

One can define a function:

```
void plot( double x0, double x1, double h,
           const std::function< double( double ) > & f )
{
   for( double x = x0; x <= x1; x += h )
   {
      std::cout << x << "    " << f(x) << "\n";
   }
}

plot( 0, 10, 1, [] ( double x ) { return x * x; } );
plot( 0, 10, 1, sq( ));
plot( 0, 10, 1, square );
```

Capture

```
struct pow
{
   unsigned int n;
   pow( unsigned int n ) : n{n} { }

   double operator( ) ( double d ) const
   {
      double res = 1;
      for( unsigned int i = 0; i < n; ++ i )
         res *= d;
      return res;
   }
};

plot( 0, 10, 1, pow(4));
```

How to make a λ from this?

```
unsigned int n = 5;
plot( 0, 10, 1, [ n ] ( double d )
   { double res = 1;
     for( unsigned int i = 0; i < n; ++ i )
        res *= d;
     return res;
   } );
```

Capture (2)

Between `[ ]`, one can write variables that the body of the lambda can use.

The variables listed between the `[ ]` become fields of the implicit object, and parameters of the constructor of the implicit object. The fields are initialized with the values of the parameters.

Capture by Value

The safest way of giving access to local variables, is to pass them by value. Write `[ v1, ... vn ]`. The constructor copies the values.

You can also write `[ = ]` . Then the compiler figures out which variables should be passed. This is easy, but not nice, because it better to make the data that are used, visible.

Capture by Reference

Parameters can also be passed by reference.

1. Copying may be inefficient for large objects. Some objects may not have a copy constructor.
2. It allows to pass information to the function object, after it has been constructed. Is this good or bad?

In the capture list, preceed reference variables with a `&`.

Sometimes, the λ may live longer than the function that constructed it. In such case, reference parameters may cause disaster!

Example of Capture by Reference

```
unsigned int n = 1;
auto func = [ &n ] ( double d )
   { double res = 1;
     for( unsigned int i = 0; i < n; ++ i )
        res *= d;
     return res;
   };

n = 5;  // Secretly modifies the meaning of func.
// There is an invisible information flow.
// Don't use C++ like Java! Stay with value semantics!

plot( 0, 10, 1, func );
```

A Lambda can surive its Reference Captures

```
std::func< double( double ) > f;

{
   unsigned int n = 1;
   f = ...  // as on previous page.
};

// Now f has survived n. Disaster!
```

The example is artificial, but this can happen in real with callbacks, or with containers.

Conclusion

Lambdas are nice, but dangerous. Use them for short-lived, local objects only.

Use capture by reference with care.