SECTION 04
Function objects

Lambdas

A lambda is a locally-defined function object (functor)

Lambdas reduce a lot of the work required in creating *ad-hoc* functor classes

The basic form of a lambda is:

```
[<Capture list>] (<Parameter list>) -> <Return type> {<Function body>}
```

Lambdas allow the programmer to define a function (strictly, a functor) locally, within block scope.

Bespoke function objects

```
class X
{
public:
  void op() { cout << "X::op()" << endl;}
};

class Functor
{
public:
  void operator() (X& elem) { elem.op(); }
};

int main()
{
  vector<X> v;
  v.push_back(X());
  v.push_back(X());
  Functor f;

  for_each(v.begin(), v.end(), f);
}
```

With STL algorithms the processing on each element is performed by a user-supplied unary or binary functor object. For common operations, the STL-supplied functors can be used (for example `std::divides`), but for bespoke manipulations a bespoke function or functor must be created.

A functor is a class that provides an implementation of `operator()`.

In the case of functors used with the STL algorithms the `operator()` function must take either one parameter (for a unary procedure) or two parameters (for binary procedures) of appropriate types.

Creating bespoke functors can be a lot of effort; especially if the functor is only used in one specific place. These bespoke functors also unnecessarily 'clutter up' the code.

A basic lambda

```
class X
{
public:
  void op() { cout << "X::op()" << endl;}
};


int main()
{
  vector<X> v;
  v.push_back(X());
  v.push_back(X());

  for_each(v.begin(), v.end(), [](X& elem) -> void { elem.op(); });
}
```

The lambda is passed each element in turn as a parameter

A lambda is defined inline, where you would normally reference a functor or call a function. The brackets ([ ]) mark the declaration of the lambda; and it should be followed by its body (the same as any other function).

Note the lambda uses a trailing return type declaration. This is (no doubt) to simplify parsing (since types are not valid function parameters)

Lambdas may have parameters

```
…
[](X& elem) { elem.op(); }
…
```

A lambda may have parameters, just like a normal function.

When the lambda is called the parameters are passed using the standard ABI mechanisms. One difference between lambdas and functions: Lambda parameters can't have defaults.

Lambda return types

```
class X
{
public:
  void op();
  int getVal();
};


int main()
{
  vector<X> v;
  v.push_back(X());
  v.push_back(X());

  for_each(v.begin(), v.end(), [](X& elem) { elem.op(); });

  auto i = find_if(v.begin(), v.end(),
                   [](X& elem)->bool
                   {
                     return (elem.getVal() != 0);
                   } );
}
```

Lambdas may return values to the caller.

Lambdas must use the trailing return type syntax.

The return type may be omitted if:

- The return type is void
- The compiler can determine the return type (lambda body is return <type>;

Block-scope functions

```
void func()
{
  auto lambda = [](int a) -> int         // MUST use auto since the lambda
                {                        // is compiler-generated and
                   return a - 1;         // therefore has an unknown type
                };

  int i = lambda(100);                   // Now you can call the lambda
                                         // just like a normal function.
}


int main()
{
  lambda(100);                           // ERROR! lambda is not in scope
}
```

A lambda has a type and can be stored. However, the type of the lambda is only known by the compiler (since it is compiler-generated), so you must use auto for declaration instances of the lambda. (You can think of the type of a lambda as a special case of pointer-to-function)

Lambdas allow ad-hoc functions to be declared at block scope (something that was illegal before) The lambda function (functor) is only available within the scope of func() in this example; unlike a function, which would have global (or file) scope.

Capturing the context

The 'context' is the set of objects in scope

'Capture' i by value

```
int main()
{
  vector<X> v;

  // Add elements to the vector…

  int i = 10;

  for_each(v.begin(), v.end(),
    [i](X& elem)
    {
       cout << elem.getVal() * i << endl;
    }
  );
}
```

lambda has a local copy, *not* the original

The context of a lambda is the set of objects that are in scope when the lambda is called. The context objects may be *captured*, then used as part of the lambda's processing.

Care must be taken because the lambda's lifetime may exceed that of its capture list.

Capturing an object by name makes a lambda-local copy of the object.

Capturing objects by reference

Capture total by reference

```
int main()
{
  vector<X> v;

  // Add elements to the vector…

  int total = 0;

  for_each(v.begin(), v.end(),
    [&total](X& elem)
    {
      total += elem.getVal();
    });

  cout << total << endl;
}
```

Capturing an object by reference allows the lambda to manipulate its context.

Be careful here, because a lambda's lifetime may exceed the lifetime of its capture list. In other words, the lambda may have a reference to an object no longer in scope!

Capturing the whole context

```
int i;
double d;
X theX;
std::vector<double> v(1000);


auto lam1 = [&]() { /* code… */ };   // Capture everything in the
                                     // context by reference.

auto lam2 = [=]() { /* code… */ };   // Capture everything in the
                                     // context by value.
```

All variables in scope can be captured (but be careful of the overheads of doing so) - the compiler must make copies of all objects (including copy constructors), or keep references for every object that is currently in scope.

Under the hood

User code

```
[&total, offset](X& elem) { total += elem.get1() + offset; }
```

Compiler generated

```
class lambda001
{
public:
  lambda001(int& t, int o) : total_(t), offset_(o) {}
  void operator() (X& elem) {total_ += elem.getVal() + offset_;}

private:
  int& total_;     // Context captured by reference
  int  offset_;    // Context captured by value
};
```

The compiler generates an ad-hoc function object for each lambda you declare. The functor name is compiler-generated (and probably won't be anything human readable)

This is why you must use auto for declaring the type of a lambda - only the compiler knows the complete type declaration.

Callable objects

A *callable object* is any object
that can be called like a function:

A member function (pointer)
A free function (pointer)
A functor
A lambda

Callable object is a generic name for any object that can be called like a function

std::function

`std::function` is a generalised pointer-to-function that can reference any *callable object*..

```
std::function <<Return Type> (<Parameter List>)>
```

`std::function` is a template class that can hold any callable object that matches its signature. `std::function` provides a consistent mechanism for storing, passing and accessing these objects.

`std::function` can be thought of as a generic pointer-to-function that can point at any callable object.

`std::function` is found in the header `<functional>`

Using std::function for call-back

```
#include <functional>

class SimpleCallback
{
public:
  SimpleCallback (std::function<void(void)> f) : callback(f) {}
  void execute();

private:
  std::function<void(void)> callback; // void (*callback)(void)
};


void SimpleCallback::execute()
{
  if (callback != nullptr)           // Is the function valid?
  {
    callback();                      // Call like a normal function
  }
}
```

`std::function` provides an overload for `operator==` (and `operator!=`) to allow it to be compared to `nullptr` (so it can act like a function-pointer)

Using std::function for call-back

With functors…

```
class Functor
{
public:
  void operator()() { cout << "Functor" << endl; }
};

int main()
{
  Functor functor;
  SimpleCallback callback(functor);
  callback.execute();
}
```

With functions…

```
void func()
{
  cout << "Free function" << endl;
}

int main()
{
  SimpleCallback callback(func);
  callback.execute();
}
```

…or with lambdas

```
int main()
{
  SimpleCallback callback([]() { cout << "Lambda" << endl; });
  callback.execute();
}
```

The same `SimpleCallback` class can be used with any callable type - functors, free functions or lambdas, without any change.

Key points

Lambdas allow the ad-hoc created of functional objects where they are needed

Lambdas allow the creation of block-scoped functions

Lambdas can interact with outside code by capturing the local context, either by value or by reference

std::function acts as a generic pointer-to-function that can point at any callable object