G52CPP
C++ Programming
Lecture 23

Dr Jason Atkin

Last Lecture

- The slicing problem
  - Copying via a base class reference or pointer can lose the sub-class part
  - The problem of neither the copy constructor nor the assignment operator being virtual

This lecture

- Some C++11 things to know
- Functors (function objects)
- Lambda functions (anonymous functors)

We have only looked at the basics

- We have covered only the basics of C++
  - None of the class libraries
  - None of the recent additions
- We could cover many more modules with content, but my hope is that by understanding the underlying principles you can work out the rest (read about it?)
- If you want to go for a job be aware of:
  - The Standard Template Library (STL)
  - C++11 (many new features)
  - Boost (often feeds features into standards)

C++11 (1)

- C++ 11 (newly-ish standardised version)
  - See page maintained by Bjarne Stroustrup: http://www.stroustrup.com/C++11FAQ.html
- Quotes from Bjarne Stroustrup:
  - "C++11 feels like a new language"
    - I agree, new and optional replacement features
  - "Currently shipping compilers (e.g. GCC C++, Clang C++, IBM C++, and Microsoft C++) already implement many C++11 features"
    - This is increasingly correct
    - See the compiler documentation for which features have been implemented and any issues

There are a LOT of new things in C++11

- You don’t need to know most of this for the exam
  - Especially not the things I only mention briefly
  - Do know functors and lambda functions though!
- Know the principles of these for job interviews
  - Probably need a different, higher level way of thinking about C++ though – further from C
- Read this: “**Ten C++11 Features Every C++ Developer Should Use”** http://www.codeproject.com/Articles/570638/Ten-Cplusplus-Features-Every-Cplusplus-Developer

The ten selected features…

- **Auto : compiler works out the type, e.g. for variable definition**
- **nullptr : Use instead of NULL, not just a (void*)0 any more**
- **Range-based for loops : for which works as a ‘foreach’**
- **Override and final : Force function being an override (avoid mistakes), or avoid overrides**
- **Strongly-typed enums : “enum class” not exported**
- **Smart pointers : see later lecture**
- **Lambdas : anonymous functions, closures, capture variables by reference or value**
- **non-member begin() and end() : globals act with containers and overloadable**
- **static_assert and type traits : assert at compile time, e.g. useful with templates**
- **Move semantics : complicated, take ownership of something (usually a temp) on assignment/copy, rather than making copy**

From http://www.stroustrup.com/C++11FAQ.html

- __cplusplus
- alignments
- attributes
- atomic operations
- auto (type deduction from initializer)
- C99 features
- enum class (scoped and strongly typed enums)
- **[[carries_dependency]]**
- copying and rethrowing exceptions
- constant expressions (generalized and guaranteed;**constexpr**)
- decltype
- control of defaults: **default** and **delete**
- control of defaults: move and copy
- delegating constructors
- Dynamic Initialization and Destruction with Concurrency
- exception propagation (preventing it; **noexcept**)
- explicit conversion operators
- extended integer types
- extern templates
- for statement; see range-for statement
- suffix return type syntax (extended function declaration syntax)
- in-class member initializers
- inherited constructors
- initializer lists (uniform and general initialization)
- Inline namespace
- lambdas
- local classes as template arguments
- long long integers (at least 64 bits)
- memory model

From http://www.stroustrup.com/C++11FAQ.html

- move semantics; see rvalue references
- narrowing (how to prevent it)
- **[[noreturn]]**
- null pointer (**nullptr**)
- override controls: **override**
- override controls: **final**
- PODs (generalized)
- range-for statement
- raw string literals
- right-angle brackets
- rvalue references
- Simple SFINAE rule
- static (compile-time) assertions (**static_assert**)
- template alias
- template typedef; see template alias
- thread-local storage (**thread_local**)
- unicode characters
- Uniform initialization syntax and semantics
- unions (generalized)
- user-defined literals
- variadic templates

- So many new features!
- Useful to know roughly what they do even if you do not know the details
- Good to show up-to-date(ish) knowledge
  - Check Boost as well

Boost

- Web page: http://www.boost.org/

Quotes from the web page (reformatted only):

- **Boost provides free peer-reviewed portable C++ source libraries.**
- We emphasize libraries that work well with the C++ Standard Library.
- Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications.
- The Boost license encourages both commercial and non-commercial use.
- We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization.
- Ten Boost libraries were included in the C++ Standards Committee's Library Technical Report (TR1) and in the new C++11 Standard.
- C++11 also includes several more Boost libraries in addition to those from TR1. More Boost libraries are proposed for TR2.

Functors and C++11 lambda functions

Lambda functions

- A lambda function is:
  - an unnamed
  - function object (functor)
  - Capable of capturing variables in scope

- So what is a function object?
  - Called functors
  - An object which is capable of acting like a function
  - i.e. ‘call’: **`objectvar( param list );`**

Simple Functors

- Functors overload the () operator
  - () operator can take a variable number of arguments, e.g.:

```
class MyFunctor1
{
public:
   MyFunctor1() = default;

   int operator() ( int x, int y )
   {
      return x + y;
   }
};
```

- Use of it looks like a function call:

```
MyFunctor1 add;
int a = add( 1,2 );
cout << a << endl;
```

Simple Functors

- Functors overload the () operator
  - () operator can take a variable number of arguments, e.g.:

```
class MyFunctor1
{
public:
   MyFunctor1() = default;

   int operator() ( int x, int y )
   {
      return x + y;
   }
};
```

Constructor: not actually needed here

Overload of operator()
You can specify the number and type of parameters. Two ints here

- Use of it looks like a function call:

```
MyFunctor1 add;
int a = add( 1,2 );
cout << a << endl;
```

1. Create object
2. Use the object – looks like **global** function call but uses operator ()

Functors with member data

```
class MyFunctor2
{
public:
   MyFunctor2( int iAdd )
      : iAdd( iAdd )
   {
   }

   int operator() ( int x )
   {
      return x + iAdd;
   }

protected:
   int iAdd;
};
```

- Could consider functors to be a function with some associated data
- We could pass information into the constructor, store it, and use it later

```
MyFunctor2 add3( 3 );
int b = add3( 1 );
cout << b << endl;
MyFunctor2 add12( 12 );
int c = add12( 1 );
cout << c << endl;
MyFunctor2 add10( 10 );
int d = add10( add10(10) );
cout << d << endl;
```

Functors with member data

```
class MyFunctor2
{
public:
   MyFunctor2( int iAdd )
      : iAdd( iAdd )
   {
   }

   int operator() ( int x )
   {
      return x + iAdd;
   }

protected:
   int iAdd;
};
```

Constructor initialises data member

Operator() uses data member

Note the data member

- Could consider functors to be a function with some associated data
- We could pass information into the constructor, store it, and use it later

```
MyFunctor2 add3( 3 );
int b = add3( 1 );
cout << b << endl;
MyFunctor2 add12( 12 );
int c = add12( 1 );
cout << c << endl;
MyFunctor2 add10( 10 );
int d = add10( add10(10) );
cout << d << endl;
```

Functors taking variables by reference

```
class MyFunctor3
{
public:
   MyFunctor3( int& var )
      : var( var )
   {
   }

   int operator() ( int x )
   {
      return x + var;
   }
protected:
   int& var;
};
```

- We could store information by reference (not a copy)

```
int iAdd = 5;
MyFunctor3 addVariable(iAdd);

cout << addVariable( 1 )
             << endl;

iAdd = 19;
cout << addVariable( 1 )
             << endl;

iAdd = -12;
cout << addVariable( 1 )
             << endl;
```

Functors taking variables by reference

```
class MyFunctor3
{
public:
   MyFunctor3( int& var )
      : var( var )
   {
   }

   int operator() ( int x )
   {
      return x + var;
   }

protected:
   int& var;
};
```

Constructor takes variable by reference and initialises data

Operator() uses data member

Note: the data member is a reference

- We could store information by reference (not a copy)

```
int iAdd = 5;
MyFunctor3 addVariable(iAdd);

cout << addVariable( 1 )
            << endl;

iAdd = 19;
cout << addVariable( 1 )
            << endl;

iAdd = -12;
cout << addVariable( 1 )
            << endl;
```

Functions (not functors!) as arguments

```
int fnadd1( int i ) { return i + 1; }
int fntake10( int i ) { return i - 10; }

void ApplyToAllElements2( int* aiArray,
      int iNumElems, int(*funcptr)(int) )
{
      for ( int i = 0; i < iNumElems; i++ )
            aiArray[i] = funcptr( aiArray[i] );
}
```

```
int aiMyArray[] = { 0,1,2,3,4,5,6 };
ApplyToAllElements2( aiMyArray, 7, fnadd1 );
for ( const int& myelem : aiMyArray )
      cout << myelem << " ";
cout << endl;
```

Functors as function arguments

```
int aiMyArray[] = { 0,1,2,3,4,5,6 };
iAdd = 20;

ApplyToAllElements( aiMyArray, 7, addVariable );
for ( const int& myelem : aiMyArray ) // range-for
       cout << myelem << " ";
cout << endl;

template < typename T >
void ApplyToAllElements( int* aiArray, int iNum, T addfunc )
{
       for ( int i = 0; i < iNum; i++ )
       {
              aiArray[i] = addfunc( aiArray[i] );
       }
}
```

Storing these for use: std::function

- **`Std::function< return type (param types) >`**
  - Template class, type specifies return ty
- E.g. : **`std::function<int(int)> f`**
  - f can store a function pointer or functor with return type int and single int parameter
- Store, copy or invoke any callable of correct type
  - E.g. function pointer or functor
- Useful if you need to store these things for calling later
- The template version of function parameters will accept function pointers or functors anyway, so std::function is not needed for parameter types
  - Unless you need to avoid a template for some reason
- Note: standard template function versions can potentially inline the function call. Using this may prevent the inlining

Next lecture

- Multiple Inheritance