Generating Serialisation Code with Clang

EURO-LLVM CONFERENCE

12th April 2012

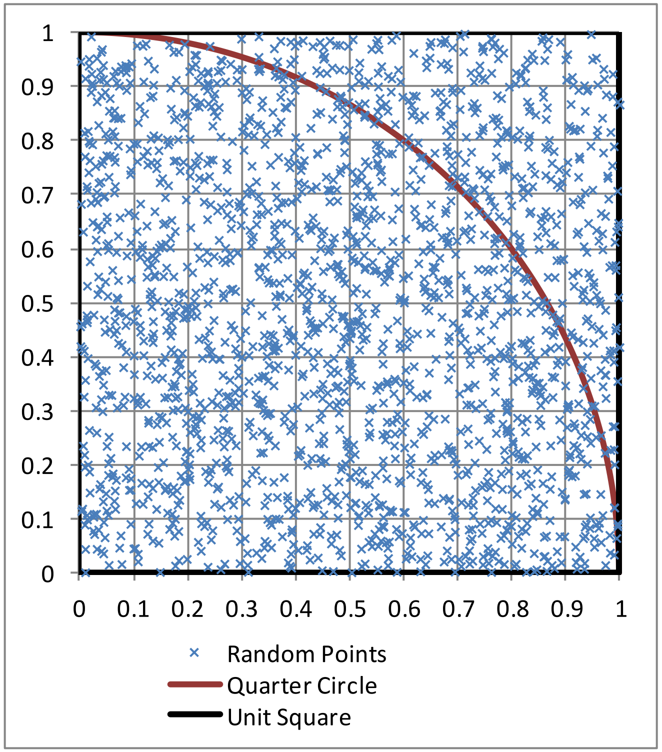
Wayne Palmer

Generating Serialisation Code with Clang
INTRODUCTION TO THE QUANTITATIVE ANALYTICS LIBRARY

- A single C++ library of nearly 10 million lines of code.
- Delivered 2-3 times a week to each of the trading and risk management desks around the bank.
- Calculates risk.
- Calculates how to hedge that risk.

Generating Serialisation Code with Clang
INTRODUCTION TO THE QUANTITATIVE ANALYTICS LIBRARY

- Market behaviour generally too complex to model using closed-form methods.
- Significant amount of calculations in QA use Monte Carlo techniques.
- Monte Carlo computationally expensive.



Generating Serialisation Code with Clang
THE NEED FOR SERIALISATION

- On a single machine, risk calculations would take days or even weeks.
- The requirement is for risk calculations to take anywhere from milliseconds to overnight.
- For speed, calculations are distributed to a grid.
- Currently there are:
  - 55,000 CPU Cores.
  - 25,000 servers and scavenged workstations.
  - 100 million tasks are run each day.
  - 50% CPU core growth each year.
- To communicate between processes on the grid we need to serialise data structures.

Generating Serialisation Code with Clang
THE NEED FOR SERIALISATION

- Hand written serialisation code.
  - Time consuming.
  - Not a scalable solution.
    - Maintenance costs increase as library size increases.
    - Fundamental changes need to be rolled out to every data structure.
  - Prone to human error.
  - Hard to enforce standards.

Generating Serialisation Code with Clang

A BIT OF HISTORY

- Automation of serialisation code generation – previously used Doxygen.
  - Not designed with the intention of generating code for serialisation.
  - Would only run on directories and not individual translation units.
  - Built up massive data structures in memory.
  - Slow.
  - Not a C++ compiler, but a C++ parser.
  - Implementation not separated from functionality.
  - Difficult to integrate with the build system.

Generating Serialisation Code with Clang
ENTER CLANG

- Serialisation with Clang
  - Clang AST easy to use.
  - Fast.
  - Accurate.
  - Can use attributes to identify structures for serialisation.
  - Generate customised errors with respect to serialisation.
  - Runs on translation units.
    - Seamlessly integrated into our Build System.
  - Can easily deliver wholesale changes to serialisation.
    - Easily rollout new output formats, i.e. JSON, XML, Binary.
    - Change existing formats.

Generating Serialisation Code with Clang
THE POWER OF ATTRIBUTES

- Clang has great support for finding attributes.
- Separates functionality from implementation.
  - Can easily add / remove serialisation by adding / removing the attribute.
  - Don’t have to alter the class implementation.
- Hard to mistake the identity of the class.

```
#ifdef __clang__
# define ATTR(...) __attribute__((annotate( " " #__VA_ARGS__ )))
#else
# define ATTR(...)
#endif
```

Generating Serialisation Code with Clang
THE POWER OF ATTRIBUTES

```
#ifndef SIMPLE_H
#define SIMPLE_H

#include "ATTR.h"

class ATTR(serialise) Simple
{
public:
   Simple() : m_SerMember(65) {}

   int m_SerMember;

   virtual void needAMethod();
   virtual ~Simple() {}
};

#endif // SIMPLE_H
```

Generating Serialisation Code with Clang
THE POWER OF ATTRIBUTES

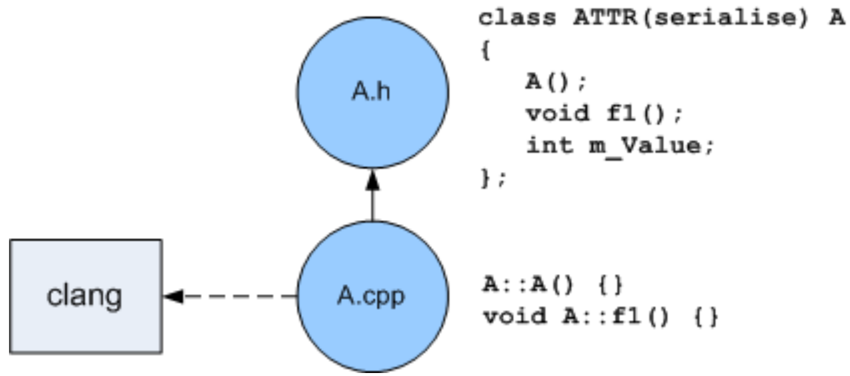
```
#ifndef SIMPLE_H
#define SIMPLE_H

#include "ATTR.h"

class ATTR(serialise) Simple
{
public:
   Simple() : m_SerMember(65), m_NoSerMember(65) {}

   int m_SerMember;
   char ATTR(no_serialise) m_NoSerMember;

   virtual void needAMethod();
   virtual ~Simple() {}
};

#endif // SIMPLE_H
```

Generating Serialisation Code with Clang
THE POWER OF ATTRIBUTES

```
#ifndef SIMPLE_H
#define SIMPLE_H

#include "ATTR.h"

class ATTR(hand_serialise(HandSer.h)) Simple
{
public:
   Simple() : m_SerMember(65) {}

   int m_SerMember;

   virtual void needAMethod();
   virtual ~Simple() {}
};

#endif // SIMPLE_H
```

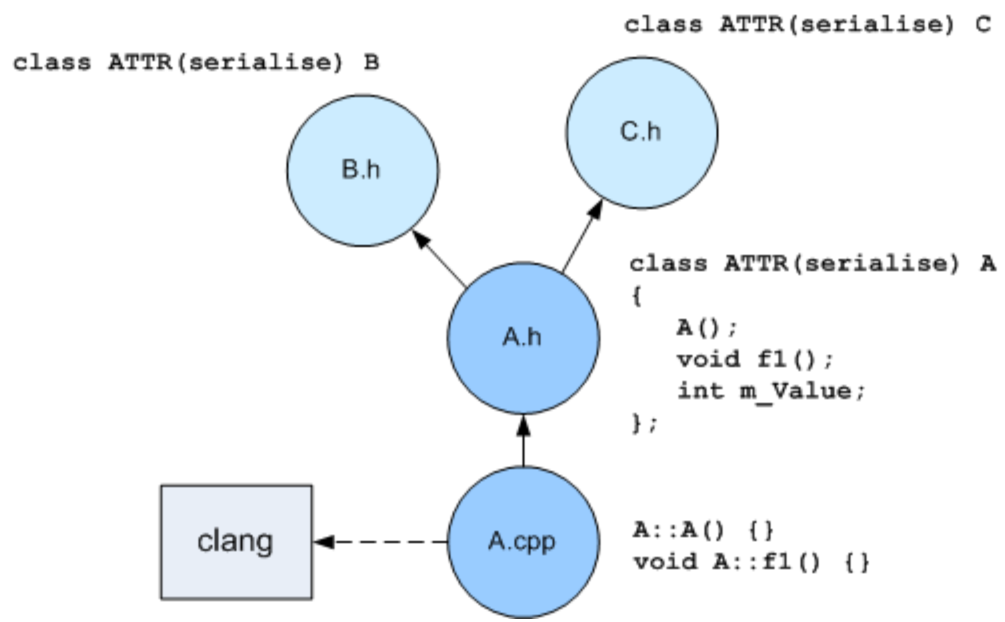
Generating Serialisation Code with Clang
ONCE AND ONLY ONCE

- How do we identify the data structures for which we want to generate serialisation code?
- How do we ensure each data structure has serialisation code generated?
- How do we ensure this is all done seamlessly within the build system?



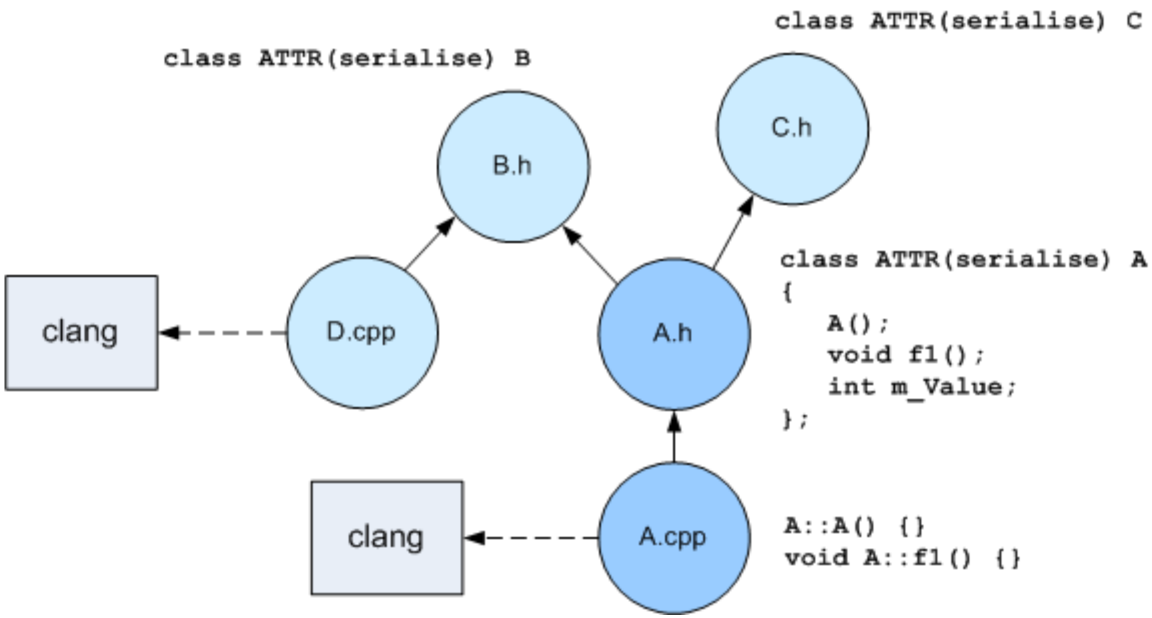
Generating Serialisation Code with Clang
ONCE AND ONLY ONCE

- Translation units can include many declarations of classes that require serialisation.



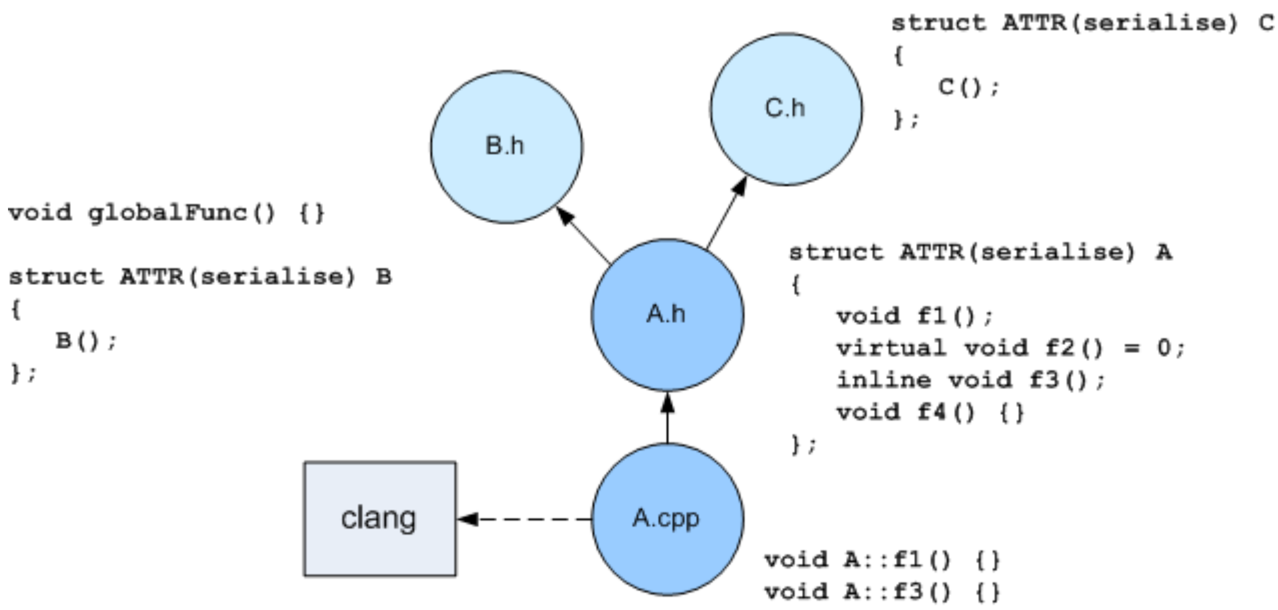
Generating Serialisation Code with Clang
ONCE AND ONLY ONCE

- Class declarations that require serialisation can be included in more than one translation unit.



Generating Serialisation Code with Clang
FINDING THE KEY FUNCTION

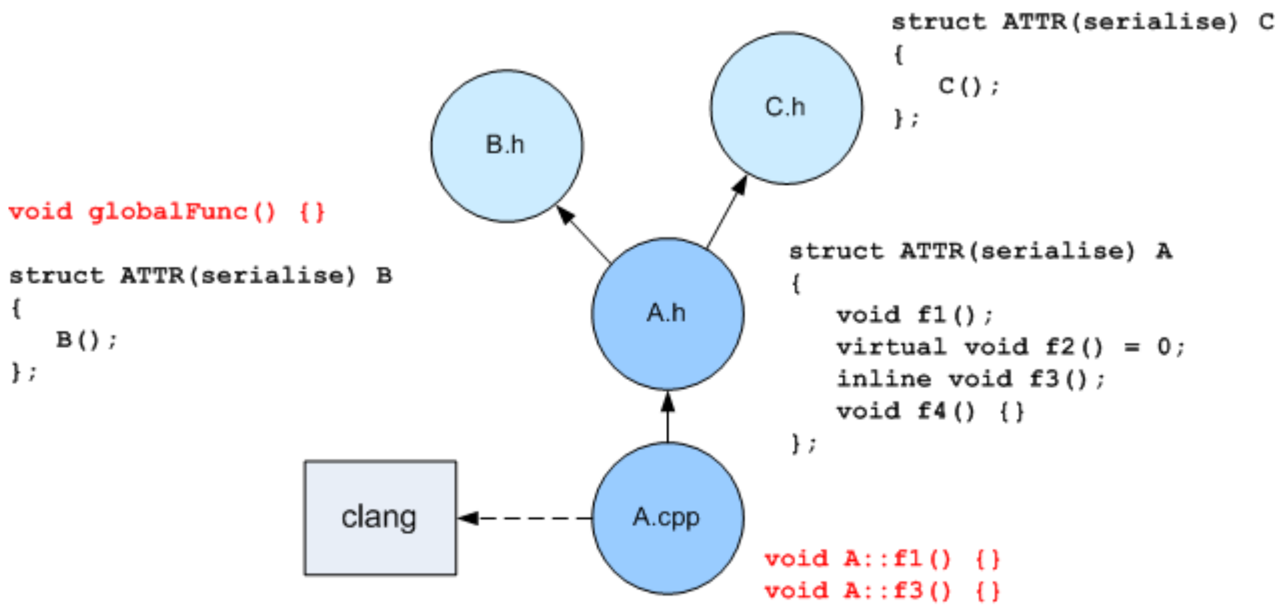
- Must find a "key function".
  - A method that makes this class unique to this translation unit.
  - Same as Clang "key function" for finding where to place a v-table.
  - However, don't care if it is virtual or non-virtual.



Generating Serialisation Code with Clang
FINDING THE KEY FUNCTION

- Visit each method of the Clang AST (CXXMethodDecl).

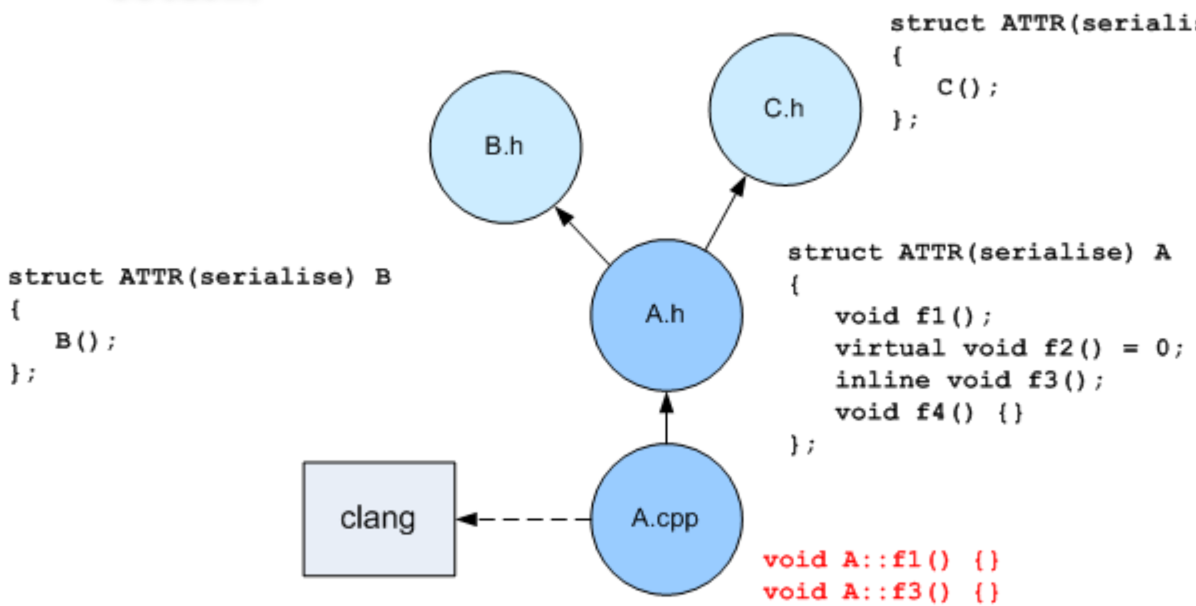
```
void Action::VisitCXXMethodDecl( CXXMethodDeclIter iter )
    { }
```



Generating Serialisation Code with Clang
FINDING THE KEY FUNCTION

- Throw away methods that have no declaration context (global scope).

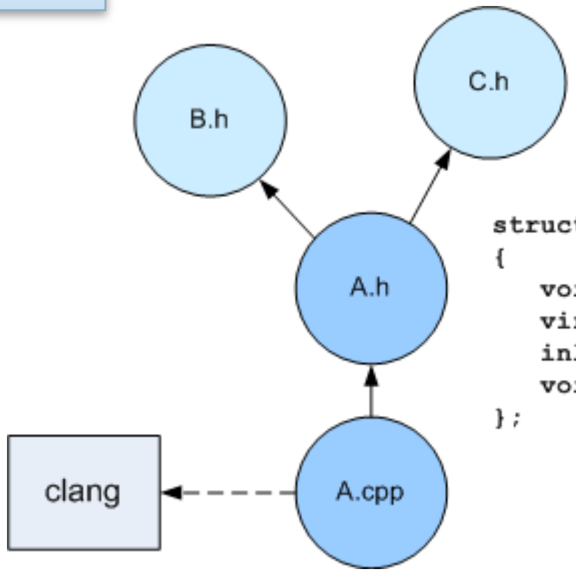
```
clang::DeclContext const * declCtxt(iter->getDeclContext());

   if (!declCtxt)
      return;
```



Generating Serialisation Code with Clang
FINDING THE KEY FUNCTION

- If the declaration context is a class or struct (CXXRecordDecl) then take a closer look at this class.
- Traverse each method of this CXXRecordDecl looking for a key method.

```
if (clang::CXXRecordDecl const * cxxRecDecl =
  dyn_cast<clang::CXXRecordDecl>(declCtxt))
```

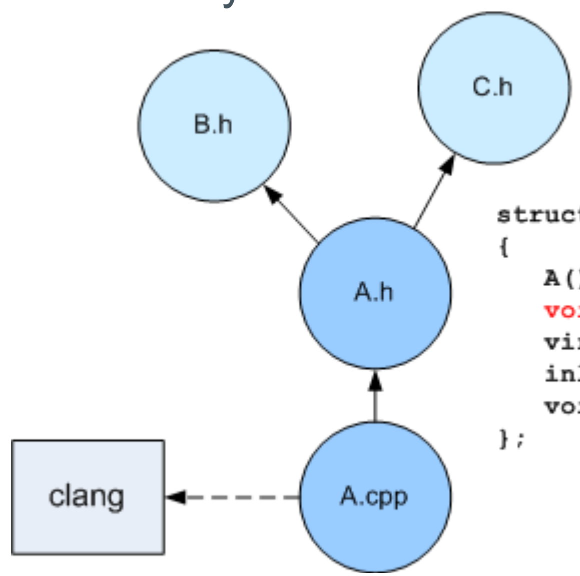



```
struct ATTR(serialise) A
{
    void f1();
    virtual void f2() = 0;
    inline void f3();
    void f4() {}
};
```

Generating Serialisation Code with Clang
FINDING THE KEY FUNCTION

- Key function won't be unique if it is in the header file, i.e.:
  - Implicitly generated by the compiler (i.e. constructors).
  - Inline specified or have an inline body.
- Pure virtual function – most probably has no implementation.
- If the function is none of these things, it is the key function.

```
if (methodDecl->isPure())
   continue;
if (methodDecl->isImplicit())
   continue;
if (methodDecl->isInlineSpecified())
   continue;
if (methodDecl->hasInlineBody())
   continue;
foundDecl = methodDecl;
break;
```




```
struct ATTR(serialise) A
{
    A() {}
    void f1();
    virtual void f2() = 0;
    inline void f3();
    void f4() {}
};
```

Generating Serialisation Code with Clang
CAN'T FIND A KEY FUNCTION

- What if a class that requires serialisation has no key function?
  - Manually add a "key method".
  - *size_t = sizeof(T)* ensures that T is a complete type.

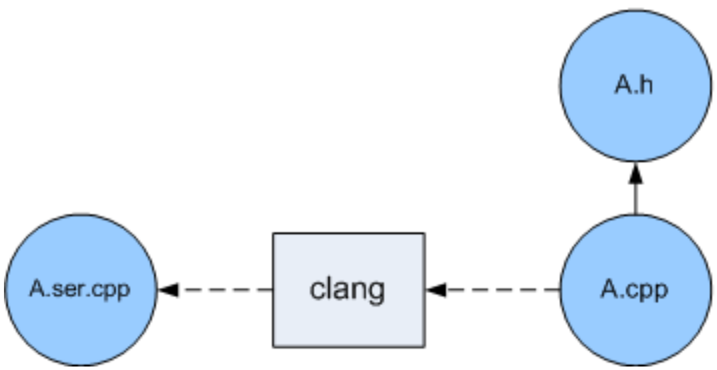
```
struct Reflection
{
  template<typename T> static void owner(const T &, const size_t =
   sizeof(T));
};
```

```
#define OWN_THE_SERIALISATION_FOR(TYPE) \
    template<> void Reflection::owner(const TYPE &, const size_t);
```

```
OWN_THE_SERIALISATION_FOR(A)
```

Generating Serialisation Code with Clang
GENERATING FOR A SIMPLE CLASS

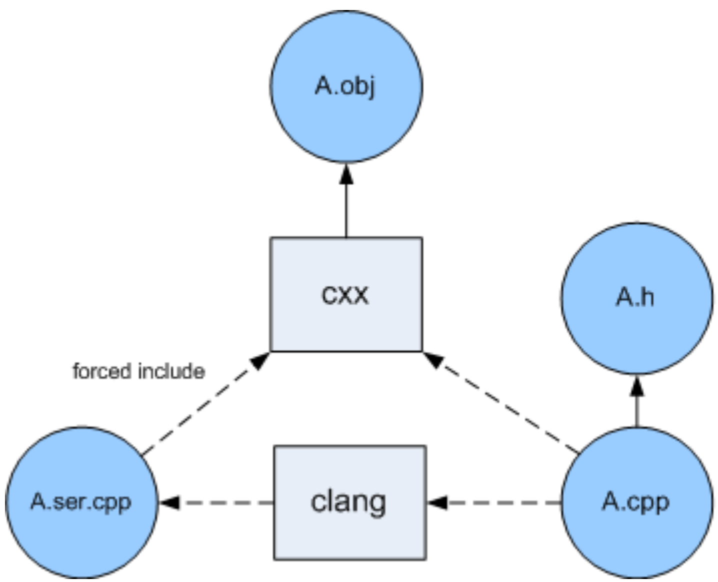
- Now that we have found a unique identifier for struct A (the key method of A), check that it has attribute "serialise".
  - If so, Clang can easily generate code capable of serialising the object in the file A.ser.cpp.




```
struct ATTR(serialise) A
{
    A() {}
    void f1();

    int m_Value;
};


A::A() : m_Value(25) {}
void f1() {}
```

Generating Serialisation Code with Clang
COMPILING FOR A SIMPLE CLASS

- The build system then "force includes" the file A.ser.cpp into A.cpp.
- Seamlessly, the developer's struct A is now capable of being serialised / de-serialised.

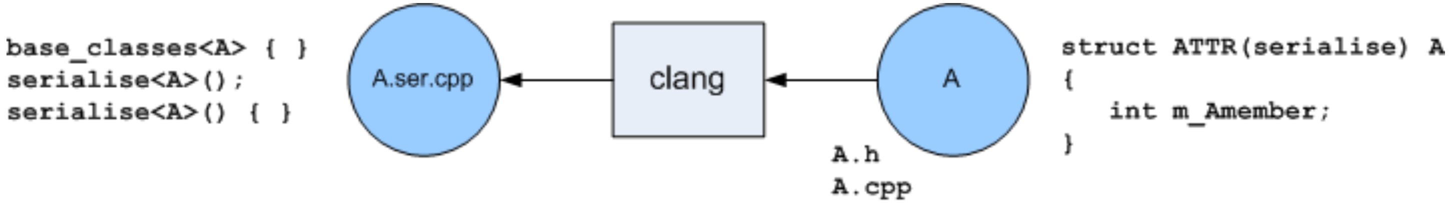



```
struct ATTR(serialise) A
{
   A() {}
   void f1();

   int m_Value;
};


A::A() : m_Value(25) {}
void f1() {}
```

Generating Serialisation Code with Clang

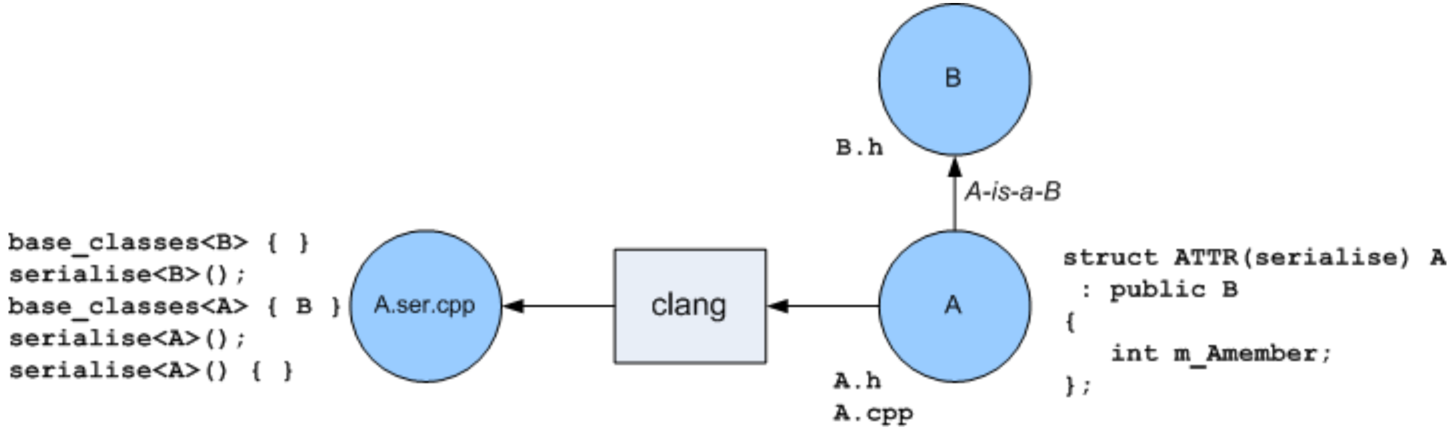
WHAT CLANG GENERATES – ONE CLASS

- Code generation for a simple struct (or class) A.
  - Generate the declaration for the serialise function.
  - Generate the definition for the serialise function.



Generating Serialisation Code with Clang
WHAT CLANG GENERATES – INHERITANCE

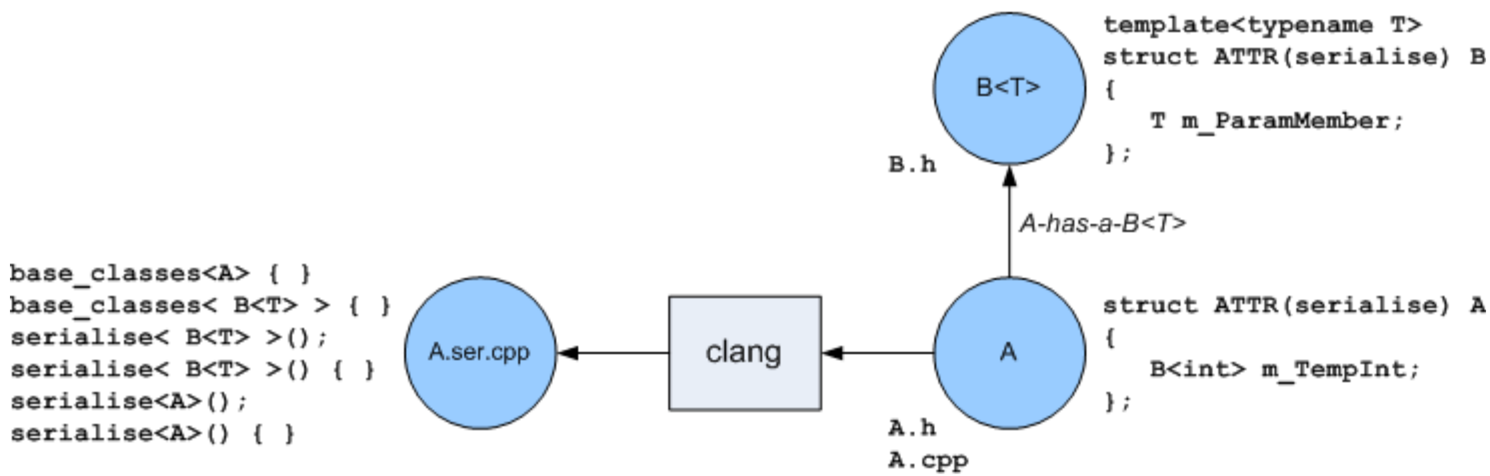
- When struct A derives from B:
  - Since we want our build system to compile this file straight away, the declaration for serialising B (*serialise<B>()*) must be generated now in the file A.ser.cpp.
    - Without this, “gates” in the build system would have to be introduced.
  - All classes that inherit from B will generate this declaration. Clang will generate the definition for *serialise<B>()* when processing B.



Generating Serialisation Code with Clang

WHAT CLANG GENERATES – IMPLICIT TEMPLATES

- For templated types we chose to generate templates rather than specialisations – less code generation required.
- The declaration and definition for templated types must be generated by Clang.



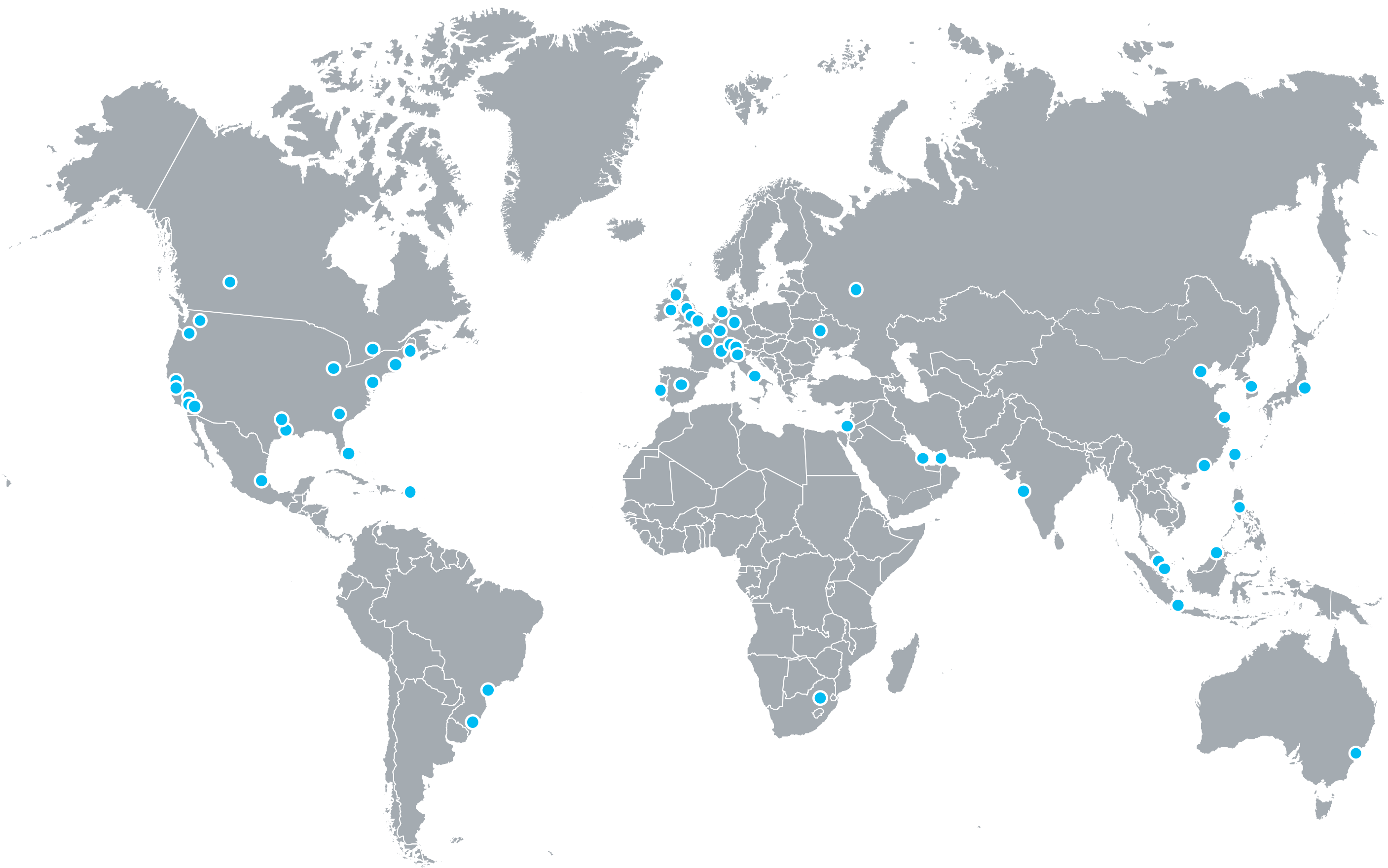
Generating Serialisation Code with Clang
OTHER USES OF CLANG WITHIN BARCLAYS

- Automatic generation of the Quantitative Analytics library interface.
  - Keyhole interface similar to COM.
  - Must maintain backwards compatibility.
  - Generates C++, COM, SWIG (Java), .NET (C++/CLR) interfaces automatically.

- Enforcing standards on the use of framework classes.

- Thread safety mark-up.

QUESTIONS



EMEA
Amsterdam
Birmingham
Doha
Dubai
Dublin
Frankfurt
Geneva
Glasgow
Johannesburg
Kiev
Lisbon
London
Lugano
Luxembourg
Madrid
Manchester
Milan
Moscow
Paris
Prague
Rome
Tel Aviv
Zurich

AMERICAS
Atlanta
Boston
Buenos Aires
Calgary
Chicago
Dallas
Houston
Los Angeles
Menlo Park
Mexico City
Miami
New York
Portland
Puerto Rico
San Diego
San Francisco
Santa Monica
Sao Paulo
Seattle
Toronto
Washington, DC

ASIA
Beijing
Hong Kong
Jakarta
Kuala Lumpur
Labuan
Manila
Mumbai
Seoul
Shanghai
Singapore
Sydney
Taipei
Tokyo