Report on the Evolution of C++

By Francis Glassborow

Overview

- Library Additions
- Core Additions
- Work of the Evolution Working Group

Library

- TR1 minus Special Math functions added
- Support for std::string as well as char*
- More Later Today

Core Working Group

- Defect Reports
- Right Angle Brackets (N1757)
- Extern Template (N1987)
- Delegating Constructors (N1986)
- Auto Proposal (N1984)

Right Angle Brackets

- Fixed embarrassment of:

```
vector<vector<int>> data;
```

Delegating Constructors

- Support delegation from constructor to another constructor of the same type.

```
class ex {
  int i;
public:
  ex(int);
  ex():ex(1){};
};
```

Auto Proposal

- Allow type deduction in declarations.

  ```
  auto value = foo(arg);
  ```

- Important for generic programming
- Problems with references and pointers

Evolution

- Big Changes
- User Support
- Small changes and cleanups

Big Changes

- Garbage Collection
- Threads
- Concurrency
- Memory Model
- Concepts
- Contract Programming

C99 and C TR Issues

- long long int
- __func__
- Consistency
- Special Math Functions
- Decimal Floating Point Types

Cleaups of C++

- Examples include:
  - Repeated template closures (i.e. > >)
  - Explicit conversion operators
  - Delegating Constructors
  - Inheriting Constructors

Explicit Conversion Operators

- The Problem: Suppressing implicit conversions
- Why? Generic code
- The Solution: allow explicit qualification

Delegating Constructors 1

- Consider:

```
int foo(int, double);
int foo(double val){return foo(0,val);}
int foo(int val) {return foo(val, 0.0);}
```

- What if foo is a constructor?

Delegating Constructors 2

- Allow:

```
class foo {
int  ival;
double dval;
public:
foo(int, double);
foo(int val):foo(val, 0.0){}
foo(double val):foo(0, val){}
};
```

Delegating Constructors 3

- The Devil is in the Details
  - –Handling exceptions
  - –When is the object alive?

Inheriting Constructors 1

-
  ```
  class derived: public base {
      // inherits almost all the members
      // base copy assignment is hidden
      // The Constructors are hidden
  }
  ```

  So what about a using declaration?

Inheriting Constructors 2

- So what about a using declaration?
  - Private constructors?
  - Access in general
  - Constructors do not have names

End User Support

- Examples include:
  - Initialiser Lists
  - Move Semantics
  - Lambda
  - Generalised Constant Expressions
  - Modules

Initialiser Lists (N1919)

- C++ already allows:

```
int i = {1};
int array[] = {1, 2, 3, 4};
```

- C++ but NOT:

```
std::vector<int> vec = {1, 2, 3, 4};
```

Initialiser Lists 2

- A new type of constructor (sequence constructor):

```
class C {
  // from a sequence of ints
  C(initializer_list<int>);
  // …
};
```

Initialiser Lists 3

- Generalise the Syntax. So:

```
std::vector<int> vec{1, 2, 3, 4};
```

- Allow it everywhere:

```
foo(std::vector<int> & const);
foo({1, 2, 3, 4});
```

Initialiser Lists 4

- Really everywhere:

```
mytype mt{}; // default initialise
mytype mt = {}; // exact equivalence
mytype mt[] = {};
```

- Really, really everywhere:

```
mytype(v);
```

Initialiser Lists 5

- Narrowing Conversions:

```
char c = 456;
char c = {456};
```

Can we make the second ill-formed?

Move Semantics

- See N1690 for details
- Optimisations
- Already Implemented (CodeWarrior)

Generalised Constant Expressions

- What
- Why
- How

A Good Starting Point

- www.open-std.org/jtc1/sc22/wg21/
- N1969 for summary up to February 2006

Modules

Another Presentation