Doc No: SC22/WG21/N1581
J16/04-0021

Date: February 13, 2004

Project: JTC1.22.32

Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Email: hsutter@microsoft.com

Francis Glassborow
Association of C & C++ Users
64 Southfield Road
Oxford UK OX4 1PA
Email: francis.glassborow@ntlworld.com

Delegating Constructors

1. The Problem and Current Workarounds

1.1. Overview

C++ does not provide a mechanism by which one constructor can delegate to another. This means that where it is not possible (or considered undesirable) to use default arguments, the class maintainer has to write and maintain multiple constructors. This can lead to tedious code duplication at both source and object level, which impedes maintainability (because of the potential for introducing inconsistencies) and in some cases can lead to minor code bloat.

Other OO languages, such as Java, do provide this feature. As Java is often used as an introductory language, an increasing number of people learn C++ with prior experience of Java. There are substantial and desirable differences between the languages without having ones for which the only justification is historical.

Today's workarounds boil down to delegating work to a common initializer function, which does not permit delegating the initialization of bases and members.

This paper proposes extensions to constructors that will reduce repetitive coding which is tedious and fragile. The proposed changes are pure extensions to ISO C++ that will not affect the meaning of existing programs.

We note that recent independent proposals from author Glassborow (see N1445) and C++/CLI (see [C++/CLI-WD1.1]) are nearly identical in syntax and semantics. This proposal is based on both of these recent works.

This proposal falls into the following categories:

- Improve support for library building.
- Make C++ easier to teach and learn, particularly in competition with languages like Java.

1.2. Current Workarounds

Today, C++ books recommend code like the following that delegates to a common initialization function:

```
class X {
  void CommonInit();
  Y y_;
  Z z_;
public:
  X();
  X( int );
  X( W );
};
```

```
X::X()         : y_(42), z_(3.14) { CommonInit(); }
X::X( int i )  : y_(i),  z_(3.14) { CommonInit(); }
X::X( W e )    : y_(53), z_( e )  { CommonInit(); }
```

This is undesirable for the following reasons:

- *Constructor body redundancy.* In this case, one of the constructors could be replaced with a default parameter. The other can't, at least not without changing the calling semantics.
- *Member initialization redundancy.* The workaround cannot delegate the actual initialization of the member variables too, at least not without significantly restructuring the class (e.g., splitting off the data members into a second class held by pointer and allocated by CommonInit()). There is no way to achieve this effect without language support, because once we're in a non-constructor member function it's too late, and the members have already been constructed. There is no way to "really" delegate everything, including the member construction.

Aside: Note that novice programmers frequently (and mistakenly) think that the delegating constructor feature already exists, because code like the following compiles, although it doesn't do what they expect:

```
class X {
  int i_;
public:
  X();
  X( int );
};

X::X() { DoSomethingObservableToThisObject(); }

X::X( int i ) : i_(i) { X(); }              // oops! compiles, but no-op
```

2. Proposal

2.1. Basic Cases

We propose that a constructor of a class type X (the "delegating constructor") may have an initializer list that invokes another constructor of the same type (the "target constructor"). That is, the delegating constructor delegates the object's initialization to another constructor, gets control back, and then optionally performs other actions as well. A delegating constructor can also be a target constructor of some other delegating constructor.

For example:

```
class X {
  int i_;
public:
  X( int i ) : i_(i) { }
  X() : X(42) { }          // i_ == 42
};
```

The following rules apply:

- Out of line definitions are allowed as usual (see example below).
- At most one other constructor may be named as the target constructor. If a sibling constructor is named in the initializer list, then the initializer list shall contain nothing else (i.e., no other base or member initializers are allowed). The target constructor is selected by overload resolution and template argument deduction, as usual.
- Any target constructor may itself delegate to yet another constructor. If there is an infinitely recursive cycle (e.g., constructor C1 delegates to another constructor C2, and C2 also delegates to C1), the behavior is undefined. No diagnostic is required, because detecting the circularity can be burdensome to detect at compile time in the general case, when constructors are defined in different translation units. As a quality of implementation issue, implementers are encouraged to diagnose violations wherever this is feasible.
- Statements in the delegating constructor body are executed following the complete execution of the target constructor. Local variables in a delegated constructor body are no longer in scope in the delegating constructor body.
- The lifetime of an object begins when all construction is successfully completed. For the purposes of [C++03] §3.8, "the constructor call has completed" means the originally invoked constructor call. (Rationale: 1. Even if a target constructor completes, an outer delegating constructor can still throw an exception, and if so the caller did not get the object that was requested. 2. This formulation also preserves the Standard C++ rule that an exception emitted from a constructor means that the object's lifetime never began.)

Example:

```
class X {
  X( int, W& );
  Y y_;
  Z z_;
public:
  X();
  X( int );
  X( W& );
};

X::X( int i, W& e ) : y_(i), z_(e)    { /*Common Init*/ }
X::X()              : X( 42, 3.14 ) { SomePostInitialization(); }
X::X( int i )       : X( i, 3.14 )  { OtherPostInitialization(); }
X::X( W& w )        : X( 53, w )    { /* no post-init */ }
```

```
X x( 21 );           // if the construction of y_ or z_ throws, X::~X is not invoked
```

Example:

```
class FullName {
  string firstName_;
  string middleName_;
  string lastName_;

public:
  FullName(string firstName, string middleName, string lastName);
  FullName(string firstName, string lastName);
  FullName(const FullName& name);
};

FullName::FullName(string firstName, string middleName, string lastName)
  : firstName_(firstName), middleName_(middleName), lastName_(lastName)
{
  // ...
}

// delegating copy constructor
FullName::FullName(const FullName& name)
  : FullName(name.firstName_, name.middleName_, name.lastName_)
{
  // ...
}

// delegating constructor
FullName::FullName(string firstName, string lastName)
  : FullName(firstName, "", lastName)
{
  // ...
}
```

Example:

```
class ex {
  ex(int =0, double = 0.0, float = 0.0, std::string = "");
  ex(int, double, std::string);
  ex(int, std::string);

private:
  int j;
  double d;
  float f;
  std::string s;
};
```

```
ex::ex(int jp, double dp, float fp, std::string sp)
  : j(jp), d(dp), f(fp), s(sp)
{
  std::string message("full ctor");
  std::cout << message <<'\';
}

ex::ex(int jp, double dp, std::string sp)
  : ex(jp, dp, 1.0, sp)
{
  std::string message("float defaulted ctor");
  std::cout << message << '\';
}

ex::ex(int jp, std::string sp)
  : ex(jp, 0.0, sp)
{
  std::string message("float & double defaulted ctor");
  std::cout << message << '\n';
}
```

In the above example, the last constructor executes as if the following had been written:

```
ex::ex(int jp, std::string sp)
  : j(jp), d(0.0), f(1.0), s(sp)
{
  {
    std::string message("full ctor");
    std::cout << message <<'\';
  }
  {
    std::string message("float defaulted ctor");
    std::cout << message << '\';
  }
  {
    std::string message("float & double defaulted ctor");
    std::cout << message << '\n';
  }
}
```

(Note that this rewrite is accurate for this example, but cases using constructor function try blocks have no direct rewrite; see §2.3.)

Note that it can make sense to delegate to a constructor that actually takes fewer arguments. For example (contributed by Roger Orr): Consider `std::fstream`. The standard lists two constructors (§27.8.1.12(1-2)):

```
basic_fstream()
```

and

```
explicit basic_fstream(const char* s, ios_base::openmode mode);
```

An implementation of the second constructor using a delegating constructor would be:

```
basic_fstream::basic_fstream( const char* s, ios_base::openmode mode)
  : basic_fstream()
{
  if(open(s, mode) == 0)
    setstate(failbit);
}
```

2.2. Constructor templates

When using constructors that are templates, deduction works as usual or the template arguments can be provided explicitly. For example:

```
class X {
  template<class T> X( T, T ) : l_( first, last ) { /*Common Init*/ }
  list<int> l_;
public:
  X( vector<short>& );
  X( deque<char>& );
};

X::X( vector<short>& v )          : X( v.begin(), v.end() ) { }
                                      // T is deduced as vector<short>::iterator

X::X( const deque<char>& d )  : X<deque<char>::iterator>( d.begin(), d.end() ) { }
                                      // T does not need to be deduced
```

2.3. Constructor function try blocks

When using constructor function try blocks, the invocation of a target constructor is treated just like any other *mem-initializer*; an exception emitted from the initializer list or body of the target constructor means that the body of the delegating constructor is never entered, and the exception can be caught by the delegating constructor's function try block if there is an appropriate handler. For example:

```
class X {
  X( Y&, int, double );
  Y y_;
  int i_;

public:
  X( double, Y );
  X( Y );
};
```

```
X::X( Y& y, int i, double d )
  try : y_( y*d ), i_(i) { cout << "X::X(Y&,int,double) body" << endl; throw 1; }
  catch(...) { cout << "X::X(Y&,int,double) catch" << endl; }      // implicit rethrow

X::X( double d, Y y )
  try : X( y, 42, d ) { cout << "X::X(double,Y) body" << endl; }
  catch(...) { cout << "X::X(double,Y) catch" << endl; }           // implicit rethrow

X::X( Y y )
  try : X( 3.14, y ) { cout << "X::X(Y) body" << endl; }
  catch(...) { cout << "X::X(Y) catch" << endl; }                  // implicit rethrow

int main() {
  X x( Y() );
}

// Output
X::X(Y&,int,double) body
X::X(Y&,int,double) catch
X::X(double,Y) catch
X::X(Y) catch
```

In the above example, the last constructor executes as if the following had been written (this is pseudocode, not legal C++:

```
X::X( Y y )
  try {
    try {
      try : y_( y*3.14 ), i_(42) { cout << "X::X(Y&,int,double) body" << endl; throw 1; }
      catch(...) { cout << "X::X(Y&,int,double) catch" << endl; throw; }
    } {
      cout << "X::X(double,Y) body" << endl;
    }
    catch(...) { cout << "X::X(double,Y) catch" << endl; throw; }
  } {
    cout << "X::X(Y) body" << endl;
  }
  catch(...) { cout << "X::X(Y) catch" << endl; throw; }
```

3. Interactions and Implementability

3.1. Interactions

The proposed feature fits well with the rest of the language, naturally extending initializer-list syntax and semantics.

For template interactions, see above.

There is no impact on code that uses the constructor; the delegation is an implementation detail.

By design, there are no effects on existing code.

3.2. Implementability

There are no known or anticipated difficulties in implementing this feature.

4. Proposed Wording

In this section, where changes are either specified by presenting changes to existing wording, ~~strike-through text~~ refers to existing text that is to be deleted, and underscored text refers to new text that is to be added.

No change to the grammar is needed.

In §3.8, change "the constructor call has completed" to "all constructors have completed".

Change §12.6.2(2) as follows:

2 Names in a *mem-initializer-id* are looked up in the scope of the constructor's class and, if not found in that scope, are looked up in the scope containing the constructor's definition. [*Note:* if the constructor's class contains a member with the same name as a direct or virtual base class of the class, a *mem-initializer-id* naming the member or base class and composed of a single identifier refers to the class member. A *mem-initializer-id* for the hidden base class may be specified using a qualified name.] Unless the *mem-initializer-id* names a nonstatic data member of the constructor's class or a direct or virtual base of that class, the *mem-initializer* is ill-formed. A *mem-initializer-list* can initialize a base class using any name that denotes that base class type. [*Example:*

```
struct A { A(); };
typedef A global_A;
struct B { };
struct C: public A, public B { C(); };
C::C(): global_A() { }          // mem-initializer for base A
```

—end example] A *mem-initializer-list* can delegate to another constructor (the *target constructor*) of the constructor's class using any name that denotes the type of the constructor's class. If a *mem-initializer-id* designates the constructor's class, it shall be the only *mem-initializer*; the constructor is a *delegating constructor*, and the constructor named in the *mem-initializer* is the *target constructor*. The target constructor is selected by overload resolution and template argument deduction. Once the target constructor's body is executed successfully without an exception being thrown, the body of the delegating constructor is executed. If a delegating constructor invokes a target constructor that in turn directly or indirectly delegates to the original delegating constructor, the program is ill-formed; however, no diagnostic is required. [*Example:*

```
class ex {
  ex(int =0, double = 0.0, float = 0.0, std::string = "");
  ex(int, double, std::string);
  ex(int, std::string);

private:
  int j;
  double d;
  float f;
  std::string s;
};

ex::ex(int jp, double dp, float fp, std::string sp)
  : j(jp), d(dp), f(fp), s(sp)
{
  std::string message("full ctor");
  std::cout << message <<'\';
}

ex::ex(int jp, double dp, std::string sp)
  : ex(jp, dp, 1.0, sp)                    // invoke target constructor
{
  std::string message("float defaulted ctor");
  std::cout << message << '\';
}

ex::ex(int jp, std::string sp)
  : ex(jp, 0.0, sp)                        // invoke target constructor
{
  std::string message("float & double defaulted ctor");
  std::cout << message << '\n';
}
```

—end example] If a *mem-initializer-id* is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the *mem-initializer* is ill-formed. [*Example:*

```
struct A { A(); };
struct B: public virtual A { };
struct C: public A, public B { C(); };
C::C(): A() { }                    // ill-formed: which A?
```

—end example] A *ctor-initializer* may initialize the member of an anonymous union that is a member of the constructor's class. If a *ctor-initializer* specifies more than one *mem-initializer* for the same member, for the same base class or for multiple members of the same union (including members of anonymous unions), the *ctor-initializer* is ill-formed.

5. References

[C++03]	*Programming Language C++* (ISO/IEC 14882:2003(E)).
[C++/CLI-WD1.1]	*C++/CLI Language Specification, Working Draft 1.1, Jan. 2004* (Ecma/TC39-TG5/2004/3).