ANSI C++ Committe Meeting, November 12-16, 1990

Brück, Dag M.

1990

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7471)/1–9/(1990)

ANSI C++ Committee Meeting
November 12–16, 1990

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
December 1990

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
December 1990

Document Number
CODEN: LUTFD2/(TFRT-7471)/1–9/(1990)

Author(s)
Dag M. Brück

Supervisor

Sponsoring organisation
ABB Automation AB
Ericsson
Televerket

Title and subtitle
ANSI C++ Committee Meeting — November 12–16, 1990

Abstract

This report describes some of the key issues at the November 1990 meeting of X3J16, the ANSI C++ comittee:

- The committee agreed on the termination model for exception handling, and resolved some issues relating to protection when an exception is thrown.
- The committee decided to do nothing with respect to "overriding." The problems that would be solved by overriding inherited functions (requiring a special language construct) can be solved with existing features in the language.
- The library working group is working on string and i/o libraries.
- New issues for the extensions working group have been scheduled.
- There are proposals for making X3J16 develop an international standard directly, instead of first developing an American standard and then transforming this into an international standard.

Key words
C++, Standardization, ANSI, X3J16

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
9

Recipient's notes

Security classification

1. Summary

X3J16 has voted in favour of the termination model for exception handling, resolved some protection issues related to exception handling, and against extending the language with new features for overriding (the problems can be solved anyway).

The extensions working group will in the future work on proposals submitted to X3J16 on topics such as contravariant return types, run-time type information, etc. The core language working group has innumerable issues to resolve (and does a good job). The libraries working group will concentrate on a new and simplifies i/o-stream library, a string library, and an interface to the ANSI C library. The environment working group is apparently doing progress, and has produced reports at a steady rate. The international working group has been quite successful in building an international network of interested people, and will get more and more important if X3J16 is chartered to develop an international standard. The real disappointment is the formal syntax working group, which I believe does very little useful work at present, but consumes a major portion of the meetings.

2. Exception handling

After lengthy discussions the committee finally voted in favour of the termination model for exception handling [Ellis and Stroustrup, 1990]. An interesting argument for termination was given by Jim Mitchell (Sun Microsystems) who implemented Mesa many years back:

- Resumption is not needed. In particular, resumption was used less and less as applications matured.
- Resumption is difficult to describe and understand.
- Resumption requires a complex double-faced protocol between a library and an application. Resumable exceptions are in this respect equivalent to passing a library a pointer to a function in the application.

Another important argument is that resumption requires an intact stack (if the user wants to resume, and there is no way to know in advance that he will not). Consequently, the "object as resource allocator" technique described by Koenig and Stroustrup [1989] will not work, and other techniques are not well suited to an environment with exceptions.

The following example will demonstrate the problem. An object is used to temporarily turn interrupts off inside a critical region of the program. The application turns the interrupts off by defining an `IntOff` object, and interrupts are turned on again because the object is destroyed when leaving the scope of the definition.

```
class IntOff {
public:
    IntOff() { /* turn off */ }
    ~IntOff() { /* turn on */ }
};

void critical()
{
```

```
        IntOff now;
        // Interrupts turned off here
        throw "error";
    }
```

Objects allocated on the stack are guaranteed to be destroyed eventually under every proposed scheme. The problem is that unless the stack is unraveled at the throw point, interrupts will be turned off while the stack is unraveled and during the execution of the exception handler. The only known alternative is to decorate all functions with "catch all" handlers. In any case, some time will be spent in the runtime system locating the handler, before unrevaling the stack.

The same problem occurs in other forms of resource allocation: locking of databases, caching of data, closing temporary files, freeing of large amounts of memory (stack and heap), etc.

Another critical issue is protection when a class object is caught by a handler for a base class of this object. Without any constraints at all, it would be possible to convert an object to its private base class, which is illegal in the current language. Instead of the very strong restrictions discussed at the previous meeting, essentially the same rules that today determine if we can create a local variable in a function will also hold for creating an exception object. In addition, a handler for a private base class will not "match" an object of a derived class.

3. Overriding

A problem with name clashes that arises when two class hierarchies are merged was described in [Brück, 1990a]. Overriding (previously known as renaming) was conceived to solve a potential problem with multiple inheritance:

```
class cowboy { virtual void draw(); };
class picture { virtual void draw(); };
class animated_cowboy : public picture, public cowboy {
  // ...
};
```

The problem is to call one of the derived functions with a pointer to one of the base classes (the draw functions are virtual). This is not possible in the example above, and overriding was designed to preserve virtualness.

```
class animated_cowboy : public picture, public cowboy {
    virtual void paint() = picture::draw;
    virtual void shoot() = cowboy::draw;
};
```

One of the fundamental questions whenever a new feature is discussed is "can this problem be solved with existing features in the language?" In the case of overriding the answer was "no" for a long time, until Doug McIlroy came up with the obvious solution — to insert an intermediate class:

```
class Xpicture : public picture {
    virtual void paint();
    void draw() { paint(); }
};
```

```
class Xcowboy : public cowboy {
    virtual void shoot();
    void draw() { shoot(); }
};

class animated_cowboy : public Xpicture, public Xcowboy {
    // no problem
};
```

This solution solves the problem at the cost of introducing two new classes when a name clash occurs. The meeting decided to do nothing, i.e., not to adopt any new syntax for overriding. A detailed discussion of the problems and several possible solutions is found in [Stroustrup, 1990a].

4. New extensions

The extension working group has now handled the two "big" issues that were part of the original charter: templates and exception handling. Future work will be initiated by proposals for extensions or issues that other working groups think are beyond their expertise. The issues listed below have been submitted to X3J16, or are expected to be submitted soon.

To facilitate rapid turnaround in this working group, a guideline for submitting proposals will be written. Work will become much easier if the original proposal contains some manual-like description of the new feature, references to related parts of the reference manual, describes the problem addressed by the new feature, considers alternative solutions and how this one will affect other parts of the language.

Contravariant return types

A virtual function redefined in a derived class must return the same type as the original virtual function in the base class. Alan Snyder (Hewlett-Packard) has proposed an extension so the redefined function can return a type different from the original return type [Snyder, 1990].

```
struct B {
    virtual B* func();
};

struct D : public B {
    D* func();
};
```

The requirement is that there must exist an implicit conversion from the redefined return type to the original one (D $\rightarrow$ B). This feature is, as far as I can tell, useful and type-safe. A potential problem is that it may constrain the implementation of function calls, and increase compiler complexity.

Run-time type information

Many existing C++ programs use some mechanism for storing type information in objects. A common use is to make safe "downcasts," i.e., casting a pointer to base class into a pointer to derived class. Having seen many

bad Simula programs that used this approach instead of using virtual functions, Stroustrup deliberately designed C++ without any builtin mechanism for determining the actual type of an object at runtime. There seems to exist legitimate needs for inquiring the type, and even Stroustrup agrees that the issue should be reconsidered.

The big issue is how much information should be stored. There exist proposals for a minimum amount of information (a typename), and proposals for "meta classes" that contain a complete description of a class object. Types are currently not first-class citizens in C++, and probably never will be.

European representation

C and C++ programs are inherently difficult to represent with non-american national character sets, because of the use of `|`, `{`, and other funny characters. The ANSI C committee invented trigraphs, so `{` could be written as `??<`. Trigraphs are of course painful to write and read, and I cannot believe anyone has ever written a C program using them.

Bjarne Stroustrup and Keld Simonsen have proposed a digraph representation for ISO C, and a revised paper deals with C++ [Stroustrup, 1990b]. The new symbols are:

`or`	`|`	
`cor`	`||`	conditional or
`and`	`&`	
`cand`	`&&`	conditional and
`xor`	`^`	
`compl`	`~`	complement
`(:`	`{`	
`:)`	`}`	
`!(`	`[`	
`)`	`]`	
`!`	`[ ]`	infix subscript operator

In my view, this proposal is practical and useful; it solves a problem for many European/African users, and has a better "look and feel" than the trigraph proposal. It does not solve the more general problem of representing national character sets (e.g., Katakana) in strings, however. Some people believe the entire discussion is meaningless because the ISO Latin-1 character set takes care of the problem.

Garbage collection.

Garbage collection will surely arrive on the table of the extensions working group; there is reason to expect a proposal from France. The working group decided not deal with this (or any other) issue until a proposal is submitted.

New keyword: inherited

A number of people have asked for some way to specify an anonymous base class in C++. Basically, you would be able to write:

```
struct B { virtual void foo(); };
struct D { void foo(); };
```

```
void D::foo()
{
    inherited::foo();       // instead of 'B::foo'
    // do some more
}
```

The advantages are a more explicit inheritance relationship, and easier modification of the class hierarchy. I like the keyword `inherited` (which is used in Mac C++), but `super` is used in SmallTalk and other languages.

The proposal for X3J16 [Brück, 1990b], is enclosed in Appendix A. It has a fairly long section on reasons for introducing `inherited`.

5. International concerns

There is a possibility that X3J16 will get promoted to develop an international standard directly, instead of first developing an American standard which eventually becomes an international standard.

Changing from type 'D' to type 'I' development will not cause any significant changes in the development of the standard, until the first Draft Standard is sent out for comments. ANSI regulations already permit foreign participation in its technical committees, and X3J16 has decided to hold one meeting per year outside USA (the first is in Lund, June 1991).

The main advantages of this change would be to incorporate comments from outside USA earlier in the standardization, and to reduce the ballotting periods. There is hope of producing an international standard earlier, and with less effort.

6. Template ambiguity

The following is an interesting (?) example of difficulties of interpretating the meaning of a template, and also demonstrates the conflicts between declarations and statements in C++ (example from Andrew Koenig).

```
int x;

template <class T>
class Y {
    void f() { T::Q (x); }
};

struct A { void Q(int); };
struct B { typedef int Q; };
```

Now, these definitions are likely to confuse many compilers. There is no way to determine what `T::Q` refers to when the template is parsed; this can not be done until template instantiation, and then (at least) two meanings are possible:

`Y<A>` `Y::f` calls function `A::Q` with the global variable `x` as an argument.

`Y<B>` `Y::f` defines a local variable `x` of type `B::Q`. The parentheses are optional, but still legal.

Difficulties in interpretation are probably inevitable with a mechanism as flexible as C++ templates. It would be interesting to see to what extent other languages with generic data types (e.g., Ada) suffer from these problems.

7. References

BRÜCK, DAG M. (1990a): "ANSI C++ Committee Meeting — July 9–13, 1990," TFRT-7459, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, DAG M. (1990b): "New keyword for C++: inherited," Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, ANSI document number X3J16/90-0086.

ELLIS, MARGRET and BJARNE STROUSTRUP (1990): *The Annotated C++ Reference Manual*, Addison-Wesley.

KOENIG, ANDREW and BJARNE STROUSTRUP (1989): "Exception Handling for C++," *Proceedings of the C++ at Work conference*, October 1989, ANSI document number X3J16/90-0042.

SNYDER, ALAN (1990): "A Proposal to Revise C++ Subtyping Rules," Hewlett-Packard, CA, USA, ANSI document number X3J16/90-0049.

STROUSTRUP, BJARNE (1990a): "Summary of Overriding Issues," AT&T Bell Laboratories, Murray Hill, NJ 07974, ANSI document number X3J16/90-0098.

STROUSTRUP, BJARNE (1990b): "A European Representation for C++," AT&T Bell Laboratories, Murray Hill, NJ 07974, ANSI document number X3J16/90-0099.

Appendix A. New keyword for C++: inherited

The following is a reprint of X3J16 document number 90-0086.

Description

The keyword **inherited** is a *qualified-class-name* that represents an anonymous base class (§5.1).

inherited :: *name*

denotes the inherited member *name*. The meaning of **inherited**::*name* is that of *name*, pretending that *name* is not defined in the derived class (§10).

```
struct A { virtual void handle(); };
struct D : A { void handle(); };

void D :: handle()
{
    A::handle();
    inherited::handle();
}
```

In this example **A**::**handle** and **inherited**::**handle** denote the same name.

When a class is derived from multiple base classes, access to **inherited**::*name* may be ambiguous, and can be resolved by qualifying with the class name instead (§10.1.1).

The dominance rule does not apply to qualified names, and consequently not to **inherited**::*name* (§10.1.1).

Motivation

Many class hierarchies are built "incrementally," by augmenting the behaviour of the base class with added functionality of the derived class. Typically, the function of the derived class calls the function of the base class, and then performs some additional operations:

```
struct A { virtual void handle(int); };
struct D : A { void handle(int); };

void D :: handle(int i)
{
    A::handle(i);
    // other stuff
}
```

The call to **handle**() must be qualified to avoid a recursive loop. The example could with the proposed extension be written as follows:

```
void D :: handle(int i)
{
    inherited::handle(i);
    // other stuff
```

```
}
```

Qualifying by the keyword `inherited` can be regarded as a generalization of qualifying by the name of a class. It solves a number of potential problems of qualifying by a class name, which is particularly important for maintaining class libraries.

Unambiguous inheritance

There is no way to tell whether `A::handle()` denotes the member of a base class or the member of some other class, without knowing the inheritance tree for class D. The use of `inherited::handle()` makes the inheritance relationship explicit, and causes an error message if `handle()` is not defined in a base class.

Changing name of base class

If the name of the base class A is changed, all occurrences of `A::handle()` must be changed too; `inherited::handle()` need not be changed. Note that the compiler can probably not detect any forgotten `A::`*name* if *name* is a data member or a static member function.

Changing base class

Changing the inheritance tree so class D is derived from B instead of A requires the same changes of `A::handle()` as changing the name of the base class.

Inserting intermediate class

Assume that we start out with class D derived from A. We then insert a new class B in the inheritance chain between A and D:

```
struct A { virtual void handle(int); };
struct B : A { void handle(int); };
struct D : B { void handle(int); };
```

Calling `A::handle()` from `D::handle()` would still be perfectly legal C++ after this change, but probably wrong anyway. On the other hand, `inherited::handle()` would now denote `B::handle()`, which I believe reflects the intentions of the programmer in most cases.

Multiple inheritance

Most class hierarchies are developed with single inheritance in mind. If we change the inheritance tree so class D is derived from both A and B, we get:

```
struct A { virtual void handle(int); };
struct B { virtual void handle(int); };
struct D : A, B { void handle(int); };

void D :: handle(int i)
{
    A::handle(i);              // unambiguous
    inherited::handle(i);      // ambiguous
}
```

In this case `A::handle()` is legal C++ and possibly wrong, just as in the previous example. Using `inherited::handle()` is ambiguous here, and causes

an error message at compile time. I think this behaviour is desirable, because it forces the person merging two class hierarchies to resolve the ambiguity. On the other hand, this example shows that `inherited` may be of more limited use with multiple inheritance.

Consequences

Programs currently using the identifier `inherited` must be edited before recompilation. Existing C++ code is otherwise still legal after the introduction of the keyword `inherited`. Existing libraries need not be recompiled.

I believe `inherited` has a small impact on the complexity of the language and on the difficulty of implementing compilers.

Although this is another feature to teach, I think `inherited` makes teaching and learning C++ easier. This is not an entirely new concept, just a generalization of qualifying names by the name of a class.

Experience

This is not a new idea, and similar mechanisms are available in other object-oriented languages, notably Object Pascal. The C++ compiler from Apple has `inherited` as described above, although its use has been restricted to solving compatibility problems with Object Pascal.

Summary

This paper proposes the extension of C++ with the keyword `inherited`, a *qualified-class-name* used to denote an inherited class member. The advantages are a clearer inheritance relationship and increased programming safety. The implementation cost is small, and the consequences for existing code minor.