X3J16/91-0141
WG21/N0074
2 December 1991

How to write a C++ language extension proposal for ANSI-X3J16/ISO-WG21

members of the X3J16 working group on extensions

1 Introduction

First, let us try to dissuade you from proposing an extension to the C++ language. C++ is already too large and complicated for our taste and there are millions of lines of C++ code "out there" that we endeavor not to break. All changes to the language must undergo tremendous consideration. Additions to it are undertaken with great trepidation. Whereever possible we prefer to see programming techniques and library functions used as alternatives to language extensions.

Many communities of programmers want to see their favorite language construct or library class propagated into C++. Unfortunately, adding useful features from diverse communities could turn C++ into a set of incoherent features. C++ is not perfect, but adding features could easily make it worse instead of better.

The aim of ANSI-X3J16/ISO-WG21 is standardization of C++, not the design of C++++. The ANSI/ISO committees are trying hard to avoid inventing a new language. Few – if any committee members – believe in "design by committee." The committee's purpose is to "Standardize existing practice." This means that any proposed extension, with only very rare exception, should be implemented somewhere. However, as noted below, that a feature is implemented and useful doesn't in itself imply that it is suitable for inclusion into C++.

Let us consider what is involved in making a proposal to the committee. A modest proposal is going to cost the 100+ active members of the ANSI committee at least a couple of hours each to review and debate with other members and colleagues. It could easily cost the equivalent six full working weeks of a senior technical person's time to consider and reject a modest proposal. A complex proposal, such as exception handling or run-time type identification costs at least ten or even a hundred times that. A successful proposal implies more work. The cost to the C++ community of implementing, documenting, and learning to use your proposal is proportionally higher. If the ANSI/ISO committees and possibly the C++ community are going to make such a substantial investment in considering your proposal, you should have a proposal worth considering. Thus, if you want to make an extension to C++, you must be prepared to make a "substantial investment" of your own time and effort to present the case to the committee. Please remember that the committee members are volunteers and that your proposal competes for time with other proposals and with the professional and private lives of the committee members.

We do understand that missing features also carry costs – that is why we consider extensions at all. However, please do realize that the time and energy of every committee member is limited and that every proposal for an extension not only carries a cost but also diverts time and energy from the committees primary activity of creating an unambiguous, comprehensible, and complete description of the language and the library.

2 Questions

Here is a list of questions that might help you to refine your suggestion and possibly present it to the committee. The list presents criteria that have been used to evaluate features for C++.

[1] Is it precise? (Can we understand what you are suggesting?) Make a clear precise statement of the change as it effects the current draft of the language reference standard.

[a] What changes to the grammar are needed?
[b] What changes to the description of the language semantics are needed?
[c] Does it fit with the rest of the language?

[2] What is the rationale for the extension? (Why do *you* want it, and why would *we* also want it?)
[a] Why is the extension needed?
[b] Who is the audience for the change?
[c] Is this a general purpose change?
[d] Does it affect one group of C++ language users more than others?
[e] Is it implementable on all reasonable hardware and systems?
[f] Is it useful on on all reasonable hardware and systems?
[g] What kind of programming and design styles does it support?
[h] What kind of programming and design styles does it prevent?
[i] What other languages (if any) provide such features?
[j] Does it ease the design, implementation, or use of libraries?

[3] Has it been implemented? (If so, has it been implemented in the exact form that you are suggesting; and if not, why can you assume that experience from "similar" implementations or other languages will carry over to the feature as proposed?)
[a] What affect does it have on a C++ implementation?
[x] compiler organization
[y] run-time support
[b] Was the implementation complete?
[c] Was the implementation used by others than the implementor(s)?

[4] What difference does the feature have on code?
[a] What does the code look like without the change?
[b] What is the affect of not doing the change?
[c] Does use of the new feature lead to demands for new support tools?

[5] What impact does the change have on efficiency and compatibility with C and existing C++?
[a] How does the change affect run-time efficiency?
[x] of code that use the new feature
[y] of code that does not use the new feature
[b] How does the change affect compile and link times?
[c] Does the change affect existing programs
[x] Must C++ code that does not use the feature be re-compiled?
[y] Does the change affect linkage to languages such as C and Fortran?
[d] Does the change affect the degree of static or dynamic checking possible for C++ programs?

[6] How easy is the change to document and teach
[a] to novices?
[b] to experts?

[7] What reasons could there be for NOT making the extension? There will be counter arguments and part of our job is to find and evaluate them so you can just as well save time by presenting a discussion.
[a] Does it affect old code that does not use the construct?
[b] Is it hard to learn?
[c] Does it lead to demands for further extensions?
[d] Does it lead to larger compilers?
[e] Does it require extensive run-time support?

[8] Are there
[a] alternative ways of providing a feature to serve the need?
[b] alternative ways of using the syntax suggested?

[c] attractive generalizations of the suggested scheme

Naturally, this list is not exhaustive. Please expand it to cover points relevant to your specific proposal and leave out points that are irrelevant.

3 Remember the Vasa!

Then, if the proposal is accepted for detailed consideration in the committee, you should be prepared to attend several ANSI/ISO committee sessions or sessions of a national C++ standards committee. There - as a member or as an observer - you could present your proposal and take part in the debate about it and other proposals. If that is not possible try to find someone on the committee who is willing to make the case on your behalf.

Please understand that most extensions to the language cannot be accepted. Do not take it personally. Frequently a problem that is solved with a language extension is also solved by suitable use of existing C++ features. Alternatively, a smaller extension to an existing feature or a library class might do the job acceptably. This was the case with a recent extension proposal for generalize overriding (see Ellis and Stroustrup: The Annotated C++ Reference Manual §10.11, pg236) that Bjarne Stroustrup made. If the designer of C++ is willing to learn and withdraw a proposal, so should you.

Please also understand that there are dozens of reasonable extensions and changes being proposed. If every extension that is reasonably well-defined, clean and general, and would make life easier for a couple of hundred or couple of thousand C++ programmers were accepted, the language would more than double in size. We do not think this would be an advantage to the C++ community.

We often remind ourselves of the good ship Vasa. It was to be the pride of the Swedish navy and was built to be the biggest and most beautiful battleship ever. Unfortunately, to accommodate enough statues and guns it underwent major redesigns and extension during construction. The result was that it only made it half way across Stockholm harbor before a gust of wind blew it over and it sank killing about 50 people. It has been raised and you can now see it in a museum in Stockholm. It is a beauty to behold - far more beautiful at the time than its unextended first design and far more beautiful today than if it had suffered the usual fate of a 17th century battle ship - but that is no consolation to its designer, builders, and intended users.

4 Sample Proposal

So you still want to make an extension? Here is an example of a proposal that almost made it. It was proposed by the ISO representative from Sweden and strongly supported by the representative from Apple. There was implementation and usage experience from both a C++ implementation and from another language. In general, we liked it. The "snag" that caused the proposers to withdraw the proposal is presented at the end.

New keyword for C++: inherited

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
Box 118, S-221 00 Lund, Sweden
E-mail: dag@control.lth.se

1. Description

The keyword inherited is a qualified-class-name that represents an anonymous base class (RM, Section 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 (RM, Section 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 (RM, Section 10.1.1).

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

2. 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.

2.1 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.

2.2 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.

2.3 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.

2.4 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.

2.5 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.

3. 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.

4. 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.

5. 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.

5 Comments on the inherited:: Proposal

Dag Brück is a member of the X3J16 working group for evaluating proposed extensions; he volunteered his proposal for this use. The proposal is well-argued and – as is the case with most proposals – there was more expertise and experience available in the committee itself. In this case the Apple representative had implemented the proposal. During the discussion we soon agreed that the proposal was free of major flaws. In particular, in contrast to earlier suggestions along this line (some as early as the discussions about multiple inheritance in 1986) it correctly dealt with the ambiguities that can arise when multiple inheritance is used. We also agreed that the

proposal was trivial to implement and would in fact be helpful to programmers.

Note that this is *not* sufficient for acceptance. We know of dozens of minor improvements like this and at least a dozen major ones. If we accepted all the language would sink under its own weight (remember the Vasa!). We will never know if this proposal would have passed, though, because at this point in the discussion, Michael Tiemann walked in and muttered something like "but we don't need that extension; we can write code like that already." When the murmur of "but of course we can't!" had died down Michael showed us how:

```
class foreman : public employee {
    typedef employee inherited;
    // ...
    void print();
};

class manager : public foreman {
    typedef foreman inherited;
    // ...
    void print();
};

void manager::print()
{
    inherited::print();
    // ...
}
```

A further discussion of this example can be found on page 205 of B.Stroustrup: *The C++ Programming Language (2nd Edition)*; Addison-Wesley 1991. What we hadn't noticed was that the re-introduction of nested classes into C++ had opened the possibility of controlling the scope and resolution of type names exactly like other names.

Given this technique we decided that our efforts was better spent on some other standards work. The benefits of `inherited::` as a built-in facility didn't sufficiently outweigh the benefits of what the programmer could do with existing features. In consequence, we decided not to make `inherited::` one of the very few extensions we could afford to accept for C++.

6 Addresses

If you still want to propose an extension please send mail (electronic mail is preferred) to one of these X3J16 members that will make sure that your proposal comes to the attention of the committee. Please make it clear if you are making an official proposal. We all get lots of casual suggestions and inquiries.

Dmitry Lenkov, HP, dmitry@hpcldnl.hp.com (X3J16 chairman). 19447 Pruneridge Ave, MS 47LE. Cupertino CA95014. USA.

Stephen D. Clamage, TauMetric, steve@taumet.com (acting X3J16 vice chairman). 8765 Fletcher Pkwy, Ste. 301. La Mesa CA91942, USA.

Bjarne Stroustrup, AT&T Bell Labs, bs@research.att.com (chairman of the X3J16 working group for extenstions). Murray Hill, NJ07974. USA.

Dag Michael Brück, Sweden, dag@Control.LTH.Se

Philippe Gautron, France, gautron@rxf.ibp.fr

Bill Gibbons, Apple, bgibbons@apple.com

Ted Goldstein, Sun, tedg@sun.com

Aron Insinga, DEC, insinga@tle.enet.dec.com

Konrad Kiefer, Siemens, kk@ztivax.siemens.com

Kim Knuttila, IBM, knuttila@torolab6.iinus1.ibm.com

Martin O'Riordan, Microsoft, martino@microsoft.com

You might also consider becomming a member of X3J16 or one of the other national standards committees. We can use all the constructive help we can get.

Here is how you become a member of X3J16: Have your organization send a letter to

```
Dan Arnold
X3 Secretariat, CBEMA
311 First St NW, Suite 500
Washington  DC  20001-2178
```

saying that they want to join the X3J16 C++ Committee. Include a brief statement of why -- the organization's interest in C++, standardization efforts, or whatever. Specify who the representative(s) will be, and the desired class(es) of membership: Principal, Alternate, or Observer.

Send a copy of the letter to

```
Steve Clamage
TauMetric Corporation
8765 Fletcher Pkwy, Ste 301
La Mesa, CA  91942
```

Include the address, phone, fax, and email address (as applicable) of each person named.

The organization may become and remain a voting member by sending a Principal or Alternate member to two of every three meetings. There are three meetings per year, at least 2 of which are in the continental US or Canada. The annual fee is currently $250, which allows the organization one Principal member and one Alternate. X3 will send an invoice for the fee. Do not send money with the letter.

The organization may have only one Principal member, and as many Alternate members as it wants, with additional fees for additional members, but only one vote may be cast per organization. The organization may have Observer members (in addition to or instead of regular members), who need not attend meetings (but who may do so), and who may not vote at meetings. The fee is the same for Observers.

Each member organization gets one copy of all the X3J16 documents as they are issued (before and after each meeting), mailed to the Principal member (or to one of the Observer members if there is no Principal member). Each person on the membership list is added to the email reflector for the committee. There are also reflectors for the various working groups on particular topics, which any member may join.