ISSN 0280–5316
ISRN LUTFD2/TFRT--7491--SE

C++ Standardization Meeting March 15-20, 1992

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
May 1992

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* INTERNAL REPORT
	Date of issue May 1992
	Document Number ISRN LUTFD2/TFRT--7491--SE
Author(s) Dag M. Brück	*Supervisor*
	Sponsoring organisation ABB Automation Ericsson Televerket

Title and subtitle

C++ Standardization Meeting — March 15-20, 1992

Abstract

This report describes some of the most important "open" issues being worked on in the C++ standardization committee before and during the March 1992 meeting:

1. Run-time type information
2. Function return type relaxation
3. Namespace control
4. Name lookup
5. Lifetime of temporaries

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280–5316		*ISBN*
Language English	*Number of pages* 7	*Recipient's notes*
Security classification		

1. Overview

The most important "open" issues being worked on in the C++ standardization committee before and during the March 1992 meeting are

- Run-time type information
- Function return type relaxation
- Namespace control
- Name lookup
- Lifetime of temporaries

There are various other issues being discussed in both the extensions working group and the library working group. The core language working group has been all busy with name lookup. The environment working group seems to have committed suicide, and the formal syntax group does not seem to do anything useful at all (most of its members have gone to core language sessions).

2. Extensions working group

Run-time type information

Run-time type information (RTTI), sometimes called dynamic type information, would enable the user to determine the "true" type of an object accessed through a pointer or a reference [Stroustrup and Lenkov, 1992]. The RTTI mechanism should serve two slightly different purposes:

- Enable checked type casts, in particular, downcasting from base class to derived class.
  ```
  Base* p = new Derived;
  ....
  Derived* q = cast<Derived>(p);
  ```
  This is one possible syntax, which looks like a template class instantiation, but there are alternatives.
- Give access to information about the object. The minimum information is the name of the actual type, but it should be possible to extend the type information arbitrarily.

RTTI is common in object-oriented languages, but does not yet exist in C++. The reason is that branching on the type instead of defining suitable virtual functions may lead to bad program design. However, there seems to be good reasons for allowing access to RTTI:

- Some operations are very awkward to express with virtual functions, for example, a function that counts the number of objects of a particular kind in a list.
- It may be impossible to add a new virtual function, e.g., to an existing library without source code.
- High-level optimization. It is possible to select a better algorithm if we know the real type of the object.
- Every major library does it without any language support. Current mechanisms are incompatible and difficult to extend, which makes it difficult to combine libraries.

- On the other hand, the function return type relaxation (see next section) will reduce the overall need for type casts in C++ programs.

The checked cast operation is generally regarded as essential. The checked pointer cast would either perform the cast if it is legal, or ruturn a nil pointer; an illegal reference cast would throw an exception. Some people have argued that the old unchecked cast should be replaced with the checked cast, using the same syntax. This would break existing C++ code, in particular, libraries that use "magic" pointer values as flags. There are also some cases with multiple inheritance where the RTTI mechanism would not have enough information to do a downcast.

I think it is essential that the application will be able to intercept illegal casts, either to print debugging information, or just to log potential errors in a program. A possible sollution is the "new handler" approach in current C++, except that no recovery should be possible for illegal casts.

Access to additional type information is more controversial, and it is yet too early to give any details about the proposal.

```
Base* p = new Derived;
....
if (typeid(p) == Typeid<Derived>())
  // do something with a derived class object
```

In this example, `typeid(p)` creates a magic token which could be used for acccessing more information about the type of the object. This token can also be compared to known type names, and `Typeid<Derived>()` is a template-like syntax for creating such a typeid object.

Another problem is to make typeids unique, in particular, in a system with dynamically loaded libraries. For that reason, the typeid cannot be a simple pointer. The best way to augment the RTTI system with application-specific information is to use maps (associative arrays) which use typeids as search keys. Consequently, typeids must be small, efficient to compare, and it should be possible to create arrays of typeids. Maybe the standard should specify a class template for tables of type information.

Function return type relaxation

A very common request is to be able to write a cloning function for a class hierarchy like this:

```
struct A {
    virtual A* clone() const;
};
struct B : public A {
    B* clone() const;  // different return type
};
```

This is an example of "covariant" return types, and it is type-safe [O'Riordan, 1991], [Bruns and Lenkov, 1992]. This has not been possible in the past, but the committee voted to include the following text:

> It is an error for a derived class function to differ from a base class virtual function in the return type only, unless the return types are either both pointers to classes or both references to classes, and the class in the original return type is an accessible base class of the class in the new return type.

The proposal only allows covariant pointer or reference return types; returning the objects themselves would require more complicated code generation, and would in any case not preserve polymorphic behaviour. This feature will reduce the need for many explicit type casts.

Namespace control

A major problem in developing large C++ applications is namespace pollution, in other words, that names (identifiers) are used for different purposes in the application. The problem becomes accute when multiple libraries are used in a single program.

A proposal to encapsulate names, e.g., function declarations and class definitions, in explicitly named namespaces has been submitted by Siemens Nixdorf [Bauche *et al.*, 1992]. The ideas are basically sound, but the proposal needs to be revised and extended. The most important issues concern the impact on name lookup and overloading; the proposed syntax also causes a syntax ambiguity. It is worth pointing out that the name of the namespace will become part of any function signatures and class names. The reason is that names must be resolved at link time as well, not just at compile time. Consequently, it will not be possible to encapsulate existing header files for a binary library.

The resolution at the March 1992 meeting was for the authors to revise the proposal document based on comments during the meeting, and to continue the discussion by e-mail reflectors.

Minor proposals

There are a number of minor extensions being analyzed. Keyword arguments (as in Ada) were proposed by Siemens Nixdorf. The concept is pretty easy to integrate with existing C++, although the rules for overloading must be revised and extended. The fundamental question is whether this feature is really needed. Keyword arguments are most useful with long parameter lists, and OOP seems to encourage a programming style with simpler functions; a plausible explanation is that objects contain much state information that need not be transferred through separate arguments. Another problem is that keyword arguments make the coupling between library and application tighter (argument names cannot be changed), and that the semantics of existing libraries, which are not designed with keyword arguments in mind, may change.

Mentor Graphics has proposed special `new` and `delete` functions for arrays. There is some discussion of how these new functions should be declared, and how they interact with the runtime system. For example, most compilers allocate extra storage to register the size of the array, and this implementation detail should not be visible to the program. During the analysis of the proposal, two new variations were suggested, so the issue is still not resolved.

I have suggested that it should become possible to overload functions that only take arguments of an enumeration type. Some operations cannot be done on enumeration variables in current C++, e.g., increment and decrement:

```
enum Status {bad, good, excellent};

Status operator ++ (Status s) throw (RangeError)
{
  switch (s) {
    case bad:  return good;
    case good: return excellent;
    default:   throw RangeError();
    }
  }
```

This is currently not possible because earlier versions of C++ allowed implicit conversion from integer types to enumerations, so overloading on enumeration could change operations on integers. The proposal will be analyzed, but there seems to be no hidden difficulties. The most serious problem is that a library that defines an operation on an enumerated type may change the interpretation of a program that relied on the enumerators being converted to integers before applying the operation.

Other questions that have to be answered are if enumerations can be replaced by classes when overloading is required, and if it is possible to define a template class for enumerations. Using templates instead of built-in enumerations has the following disadvantages:

- There is no way to restrict the domain of enumerators; the user can always create another enumerator and append it to an existing enumeration. This means that a piece of code cannot assume a fixed set of enumerators.
- Enumerations with templates are somewhat awkward to declare.
- Current implementations cannot generate as efficient code for a template class implementation of enumerations. Built-in enumerations are essentially integer types with additional type constraints.

The basic requirement, to provide overloading on a set of named symbols, can be met with a template class.

The "operator dot" proposal from Jim Adcock [Adcock, 1991] has not progressed any further. Andrew Koenig and Bjarne Stroustrup have written a paper with major objections [Koenig and Stroustrup, 1991], but the second analysis paper that was delivered at the March meeting (author shall remain unnamed) turned out to be completely useless.

Philippe Gautron *et al.* have surveyed various unrelated extension and restriction proposals for templates. The issues are complicated, but a few trivial extensions, such as the ones described in [Lippman, 1991], are not unreasonable. The presentation was rather bewildering, and the audience was confused.

Rejected proposals

The extensions working group has analyzed and rejected a few proposals at the March 1992 meeting. One proposal pointed out an encapsulation loophole: that you can call the base class version of a virtual function although it has been redefined in a derived class. The proposal actually called for a different protection mechanism with finer granularity than the present scheme in C++, so it was regarded as outside the scope of the committee.

Another proposal suggested that classes should have free access to all data members, regardless of their protection. This idea breaks every form of encapsulation, and the proposal was quickly dismissed.

The third rejection concerned "invisible" private parts of a C++ class. The solution in present C++ is to use an abstract base class as the public interface, or to use a pointer to an auxiliary data structure. Both methods provide the desired functionality without language extension.

The extension working group has rejected two proposals at previous meetings: keyword inherited (can be solved with a typedef), and overriding of member function (can be done with simple class derivations).

3. Core language working group

Name lookup

The core language working group has finally made a break-through with name-lookup. The committee voted the following principles into the working document:

1. The scope of a name declared in a class consists not only of the text following the name declarator, but also of all function bodies, default arguments, and constructor initializers in that class (including such things in nested classes).
2. A name N used in a class S must refer to the same declaration when re-evaluated in its context and in the completed scope of S. *(Completed scope: the completely parsed scope, e.g., at the closing brace of the outermost enclosing class definition.)*
3. If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program's meaning is undefined.

These principles are reasonably clear and also manage to cover the obscure and pathological cases. The working group has tried to find algorithmic descriptions of earlier attempts to define name-lookup, but the invented algorithms have either been wrong or prohibitively slow, e.g., NP-complete or $O(n!)$, where n is the number of names in a class. The current principles are supposed to implementable.

Most pathological cases are the result of obscure typedefs, extra parentethes, "implicit int" or a combination thereof. Here is an illegal example:

```
typedef int T;
struct X {
    T f();
    typedef float T;
};
```

In this case typename T is redefined after it has been used as the return type of a member function. The use of "implicit int" causes many complications, for example:

```
struct Y {
    void f(const T);
    typedef int T;
};
```

When the compiler sees `const T` it could be interpreted either as an unnamed parameter of type constant T, or as the parameter T of type constant integer. The typedef defines T as a type, but we then break Principle 2. The following example is particularly obscure, but breaks Principle 3:

```
typedef int P(), Q();
struct X {
    static P(Q);
    static Q(P);
};
```

Here is an explanation by Andrew Koenig:

> The question is: for each declaration, what does it declare and what is its type?
>
> As it stands, the first declaration inside `X` declares `Q` as a static member of type `P`, which is equivalent to type `(int())`. In other

words, it declares `Q` as a member function taking no arguments and returning `int`.

Since `Q` is now defined and is not a type, the second declaration declares `Q` as a function implicitly returning `int` and accepting an argument of type `P`. This is a legitimate overloading of `Q`!

If you swap the two declarations, they now both declare `P` instead of `Q` (proof by symmetry).

The use of "implicit int" is a heritage from C. Banning this sloppy feature, or at least declaring it as an anachronism, would simplify the language specification in many cases. It would also break many existing C++ programs, which typically use this reasonably harmless construct:

```
const SIZE = 10;
```

A proposal to ban "implicit int" is not entirely unlikely, but indentifying and analyzing all the effects is a daunting task.

Lifetime of temporaries

The lifetime of temporary variables is another currently unspecified issue that must be straightened out [Koenig, 1992]. There are several possible alternatives:

1. Early destruction, i.e., essentially as soon as the temporary value has been used once. The problem is that some operations return a pointer to some internal part of the object, e.g., `String::operator const char* ()`.
2. Late destruction, i.e., at the end of the enclosing block or function. The disadvantage of this approach is that temporaries live for a long time. In the case of large objects, for example, matrices, this may cause the program to run out of free store.
3. Destruction at end of greatest enclosing expression, typically the enclosing statement. This model seems to be a reasonable compromise, although explicit pointers to internal structures are still a problem.
4. Destruction at the next branching point, i.e., a control structure or a label. This is the model IBM uses in their compiler.
5. Some other alternatives that have the drawback that it is impossible to specify when a particular implementation will actually destroy the temporaries. Consequently, it is impossible to write a program that is guaranteed to execute correctly on all implementations.

The third and fourth alternatives are the most reasonable ones, and will hopefully be voted into the working draft at the next meeting.

4. References

ADCOCK, J. L. (1991): "Request for consideration: Overloadable unary operator.()." Technical report, Microsoft. ANSI document number X3J16/91-0140.

BAUCHE, V., R. HARTINGER, and E. UNRUH (1992): "A proposal solving the name space pollution problem in C++." Technical report, Siemens Nixdorf Informationssysteme AG. ANSI document number X3J16/92-0008.

BRUNS, J. and D. LENKOV (1992): "Extending C++ to allow restricted return types on virtual functions (with addendum)." Technical report, Chicago Research and Trading. ANSI document number X3J16/92-0004.

Koenig, A. (1992): "Lifetime of temporaries." Technical report, AT&T Bell Laboratories. ANSI document number X3J16/92-0020.

Koenig, A. and B. Stroustrup (1991): "Analysis of overloaded `operator.()`." Technical report, AT&T Bell Laboratories. ANSI document number X3J16/91-0121.

Lippman, S. B. (1991): *C++ Primer*. Addison-Wesley, second edition.

O'Riordan, M. (1991): "Polymorphic over-riding of function return types." Technical report, Microsoft Corporation, Redmond, WA, USA. ANSI document number X3J16/91-0051.

Stroustrup, B. and D. Lenkov (1992): "Run-time type identification in C++." Technical report, AT&T Bell Laboratories. ANSI document number X3J16/92-0028.