ANSI C++ Committee Meetings March 11–15 and June 17–21, 1991

Brück, Dag M.

1991

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

CODEN: LUTFD2/(TFRT-7481)/1–10/(1991)

ANSI C++ Committee Meetings
March 11–15, 1991
June 17–21, 1991

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
September 1991

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

Document name
INTERNAL REPORT

Date of issue
September 1991

Document Number
CODEN: LUTFD2/(TFRT-7481)/1–10/(1991)

Author(s)
Dag M. Brück

Supervisor

Sponsoring organisation
ABB Automation AB
Ericsson
Televerket

Title and subtitle
ANSI C++ Committee Meetings — March 11–15, 1991 and June 17–21, 1991

Abstract

This report describes some of the key issues discussed during the March 1991 and June 1991 meetings of X3J16, the ANSI C++ committee.

- Type identification. Dynamic identification of an object's type at run-time, through a combination of language extensions and library support.
- Standard libraries. The string and iostream libraries are closer to final approval.
- Concurrency. A proposal has been submitted by University of Waterloo, which raises the question of language extension vs. library extension.
- Name lookup. The current language definition needs a much stricter definition of name lookup; this can of worms has just recently been opened.

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
10

Recipient's notes

Security classification

1. Type identification

Many applications and class libraries require a mechanism for run-time type identification and access to type information. Hewlett-Packard has proposed a general type identification mechanism consisting of language extensions and library support. This section is essentially a summary of [Lenkov, 1991]. The implementation strategy is outlined in [Lenkov *et al.*, 1991].

There are several reasons why type identification and access to type information is needed:

Accessing derived class functionality. Many commonly available class library consist of an inheritance hierarchy with a root class, e.g., "Object" in NIHCL. A common operation is to cast a pointer from the root class to a derived class so that a derived class member function may be invoked. This "downcast" operation is the most common case where programmers currently have to explicitly disable the C++ type system.

Exception handling. Every useful exception handling mechanism requires type identification at run-time in order to match the thrown object with the correct `catch` clause.

Accessing type information. There are some class-specific actions that are difficult to realize with the normal virtual function mechanism, for example, counting the number of objects of a particular derived type in a heterogeneous list. □

The proposal suggests a special operation to perform downcasts: `ptr_cast(A, p)`, which returns the value of pointer `p` cast to a pointer to class `A`; `p` must point to an `A` or some object derived from `A`. If `p` does not point to such an object, `ptr_cast` returns a null pointer. There is an analogous operation for references: `ref_cast(A, r)`, which either returns a reference or throws an exception. It can also be noted that the standardization committee has decided to outlaw null references (`&r == 0`).

```
List* p = // some list

if (SortedList* sp = ptr_cast(SortedList, p)) {
    Key k = sp->LastKey();
    // ....
}
```

This is an example of a construct which I think will be common in the future. The pointer `sp` is a local variable of the `if`-statement, and can only be accessed if the pointer cast succeeds.

The `stype` and `dtype` operators return a type identification for an expression that evaluates to a pointer or a reference to some object: `stype` returns the static (declared) type, and `dtype` returns the dynamic (actual) type of the expression. For example, if

```
Base* p = new Derived;
```

`stype(p)` is class Base and `dtype(p)` is class Derived.

An unresolved issue so far is what kind of information should be stored about a class at runtime. Suggestions range from "nothing" to a complete modifiable type system. The current H-P proposal talks about the name and size of the class, and a function to retrieve the type information of the base classes. It should also be possible to augment the basic type information in

some way, for example, by deriving system specific classes from a basic type information class.

Many of these operation, in particular `ptr_cast` and `ref_cast` are less meaningful for classes without virtual functions. The proposal suggests restricting the implementation of dynamic type identification to polymorphic classes (classes with virtual functions), which may be a reasonable constraint with current implementations.

2. Standard libraries

Two major libraries will most likely be included in the C++ standard: strings and iostreams. The string library may be voted on at the November 1991 meeting in Dallas. A standard concurrency library, if there will be any, is far in the future.

The string library currently being discussed [Insinga, 1991] is similar to all other string libraries, except that there is no binary string concatenation operator, `operator + (const string &, const string &)`, because of the dangers of temporary objects. Exceptions are used to signal error conditions. The proposal also contains class `w_string` which is a string of `w_char` (wide character) instead of plain `char`. The issue of collating sequences for national character sets seems to be completely overlooked.

The design of the new iostream library does not seem to have progressed as far as the string library. It is clear that there will be substantial changes compared to the current AT&T iostream library. Code that only outputs objects on streams need not change, but any code that manipulates iostreams or define special stream buffers will most likely need changes. A fundamental difference is that multiple inheritance will *not* be used in the iostream class hierarchy. Exceptions will be be defined, but it will probably be possible to force the iostream library to return status codes instead of throwing exceptions.

3. Concurrency

Some sort of standard C++ concurrency library is desirable, although there are several opinions of what it should look like. The preliminary proposal from University of Waterloo [Buhr *et al.*, 1991] is quite interesting. There are two drawbacks of the proposal that I believe will prevent it from being accepted:

- The proposal relies on substantial extensions of the basic C++ language, both new syntax and new semantics. Most importantly, the changes will have an impact on all C++ programs, not only those that use concurrency. The increase in complexity also affects every implementation.
- The proposal is written with one particular view of concurrency in mind. Although I think the proposal is reasonable, it will surely not correspond to every interpretation of the word concurrency. Because the proposal is based on language extensions and not only library extensions, there is no obvious way to substitute a different concurrency scheme.

The analysis in the paper provides valuable input to the discussion of C++ concurrency, however. It is also clear that better designs and implementations of some constructs can be provided if the language is revised without compatibility constraints.

Enter/leave functions

A fundamental problem in implementing a monitor, or some other protection mechanism, is to *guarantee* that the resource is protected. In current C++ real-time systems the programmer must explicitly use the correct mechanism, for example, wait/signal on a semaphore. It is possible to provide general enter/leave functions in a base class, but it is not possible to guarantee that they will be called. An extended version of C++ that provides similar features (and much more) is described in [Seliger, 1990].

Extending the language with proper enter/leave functions will probably not affect existing C++ programs and implementations, and will not introduce any cost when not used. A possible syntax, deliberately designed to be in line with C++ style, would be:

```
class Foo {
public:
    Foo();
    ~Foo();

    +Foo();              // Enter
    +Foo() const;

    -Foo();              // Leave
    -Foo() const;

    void f();
    void g() const;
};
```

The enter function is automatically called as the first action of calling a member function of Foo, and the leave function is called just before returning. Overloading on `const` is made according to constness of the function that was called.

```
void Foo::f()
{
    Foo::+Foo();
    ....
    Foo::-Foo();
}
```

```
void Foo::g() const
{
    Foo::+Foo() const;
    ....
    Foo::-Foo() const;
}
```

A few notes are probably in order:

- I do not see any particular need for calling the enter/leave functions for constructors or destructors. We can in any case not call `+Foo()` on an unconstructed object or `-Foo()` on a destroyed object.
- Enter/leave is inherently related to an object, so `+Foo()` and `-Foo()` should not be automatically invoked for static member functions, and maybe not for friend functions.
- Enter/leave functions are inherited just as any other member function, so `Derived::f()` calls `Base::+Base()` if no `Derived::+Derived()` exists.
- Enter/leave functions can be called using the full name `Foo::+Foo()` or `Foo::-Foo()`. The reason is that it should be possible to call the enter/leave functions of a base class, and friend functions may want to use the same protocol as member functions. In any case, the enter/leave

functions ought to be protected or private.

- Invoking `Derived::+Derived()` does not imply automatic invocation of `Base::+Base()`, but an implementation of `Derived::+Derived()` may explicitly call `Base::+Base()`, of course.

Enter/leave functions are fundamental building blocks in concurrent applications and other applications using resource protection. It is not an intrusive language extension, and may for that reason stand a chance of survival in the standardization process. It is worth noting that the immediate predecessor to C++, called "C with Classes," had enter/leave functions that nobody appreciated.

Two-phase construction

Another problem we have encountered is two-phase construction, i.e., the need to let the constructor of a base class do some additional initialization after the object is constructed. In the following example we want to make a process eligible for scheduling when it has been completely constructed.

```
class Process {
    char* stack;
protected:
    Process(int stacksize);
    virtual ~Process();
    virtual void Main() = 0;
};

extern void Schedule(Process *p);
extern void Suspend(Process *p);

Process::Process(int stacksize) : stack(new char[stacksize])
{
    Schedule(this);
}

Process::~Process()
{
    Suspend(this);
    delete [] stack;
}
```

A problem is that the object is not completely constructed when the process is made ready to run. In a typical implementation, the "vtbl" does not contain a valid entry for the virtual `Main` function. In the destructor case, the derived part of the process object has been destroyed before the base class destructor suspends the process.

These problems can of course be "solved" by the application programmer; the correct calls to schedule/suspend the process must be inserted in the constructors/destructors of the derived classes, which means that all derived classes need intimate knowledge of the process implementation. It is essential that only leaf classes call schedule/suspend, so complete knowledge of the inheritance hierarchy is also needed.

One possible solution is to re-invent the `inner` concept of Simula-67 [Birtwistle *et al.*, 1973]. Instead of using a new keyword, we can reuse `virtual`:

```
class Process {
    char* stack;
protected:
    Process(int stacksize) virtual;
    // ....
};

Process::Process(int stacksize) virtual
    : stack(new char[stacksize])
{
    virtual;
    Schedule(this);
}
```

Construction of the base class is done in two phases. Firstly, the base class constructor is run up to the `virtual` statement. Secondly, derived classes are constructed. Thirdly, the rest of the base class constructor is run. In this example, the process is not scheduled until construction (process initialization) is completed. The main advantage is that the desired behaviour can be guaranteed by the base class implementor.

Destruction can be handled by a non-member function that suspends the process and then deletes it.

```
Process::~Process()
{
    delete [] stack;
}

void KillProcess(Process* p)
{
    Suspend(p);
    delete p;
}
```

A major drawback of this scheme is that it can only handle processes allocated on the free store with `new`. Automatic process variables can no longer be used, which means that clean-up in the presence of exceptions becomes more difficult. The "inner" concept can also be applied to destructors, but the natural interpretation (to start with the derived class destructor) is not suitable for handling process suspension.

Two-phase construction is probably a more specialized extension than enter/leave functions, and consequently less likely to be standardized. It would be very interesting to see if someone has achieved the desired behaviour without language extension, except for separating construction and secondary initialization into two function calls.

4. Name lookup

Name lookup refers to the problem of finding the correct name (of a function, variable, type, etc.) given the scope rules of C++. The issue is further complicated by overloading of functions, template types, and because declaration and definition may be separated. The following example demonstrates one of the simplest cases:

```
void f(double);

struct X {
    void g();
};

void f(int);

void X::g() {
    f(3);
}
```

The question is what `f()` is called, i.e., if name lookup is made from the point of declaration or the point of definition. A slightly more mind-boggling example is:

```
struct A {
    int x;
    void f();
};

struct B {
    int x;
    friend void A::f() { x = 1; }
};
```

It is legal (even if it shouldn't be), but few compilers will get it right. The working group has found seven alternatives for the scope of a name in friend declarations.

According to [Turner, 1991], the working group is approaching consensus on resolving names in the normal cases, in which functions and classes are defined either in the scope in which they are declared, or at global scope. William Miller has proposed that most of the cases of disagreement be disallowed, i.e., member function defined as friend (the example above), and nested class defined in a block distinct from the containing class.

5. Other issues

This section describes some issues of lesser importance that are under investigation by the standardization committee.

New keyword: inherited

At the March meeting I presented a proposal to include a new reserved word, `inherited`, which in a derived class would serve as an alias for the base class. The advantage is that changes to the class hierarchy can be made without any need to change the entire source code. The proposal demonstrated several advantages of `inherited` over the current mechanism of explicitly named base classes [Brück, 1990].

Michael Tiemann (author of GNU C++) found an alternate way of accomplishing almost the same behaviour with existing language elements. The solution relies on a local `typedef` in every derived class:

```
class Derived : public Base {
```

```
    typedef Base inherited;
public:
    // ....
};
```

The type definition creates a local alias for class Base, which can be used in the derived class. The person who changes the base class must also change the `typedef`, but this is a reasonable task. There is no need to locate references to the base class throughout the source code.

The typedef-based solution has almost all advantages of using a keyword, at little added inconvenience; the original proposal was consequently turned down. The vote was unanimous.

Polymorphic overriding of function return types

A common request is to be able redefine the return type of a virtual function in a derived class, for example:

```
class Base {
   virtual Base* Next() const;
};

class Derived : public Base {
   Derived* Next() const;          // illegal
};
```

In this example the return type of the derived function is derived from the return type of the base function, which is called polymorphic overriding of the return type.

The basic problem is that the return type is not part of the function's signature and cannot be used for overloading. It turns out that polymorphic overriding is type safe and can be implemented without unreasonable overhead [O'Riordan, 1991b]. The committee has not yet decided whether to include polymorphic overriding or not.

Compile time constants in class scope

The following piece of code is not legal C++, and there is currently no good alternative.

```
class Foo {
    static const int maxSize = 100;
    char localBuffer[maxSize];
}
```

Initialization of static members must be made outside the class declaration, and at that time it is too late to set the dimension of the array. Using an enumeration constant is both inelegant and unsafe [Gibbons and Goldsmith, 1991].

There is in fact little reason not to allow the case shown above, and it can be done without complicating the language. The main unresolved issue is whether a file scope definition of `maxSize` is really needed, or if it can be treated as a compile-time constant. The proposal can also be extended to non-static members and initialization with general expressions.

Name space pollution

There are so many global names in a large application that the risk of conflict is serious. In particular, there is a risk that third-party libraries, which are not under the control of the user, introduce conflicts. The class concept is a powerful program structuring mechanism, but it does not replace a module facility as found in Modula-2 or Ada. A common approach in C and C++ is to require every vendor to choose a unique prefix for all data types and functions, but this approach is of course unmanageable in the long run.

C++ really needs a useful module facility. The current proposal from Microsoft [Rowe, 1991] is incomplete and inferior to the solutions in Modula-2 and Ada; for example, there is no way to selectively import or export names of a module. An improved proposal is expected [O'Riordan, 1991a], but the issue has not yet been seriously discussed in the extensions working group.

ISO committee

The ISO C++ committee (WG21) held its first meeting in Lund, June 18–19, 1991. The main issue was the relationship between WG21 and X3J16, and the division of labour between the two committees.

All technical discussions will in the future be conducted during joint sessions of WG21 and X3J16, chaired by the X3J16 chairman. X3J16 has adopted type I ("international") development which is suitable for developing an international standard. There is apparently reason to believe that there will be one common standard in the future.

References

Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard (1973): *SIMULA BEGIN*. Auerbach Publishers Inc., Philadelphia, PA, USA. Also published by Studentlitteratur.

Brück, D. M. (1990): "New keyword for C++: inherited." Technical report, Dept. Automatic Control, Lund Inst. of Technology, Lund, Sweden. ANSI document number X3J16/90-0086.

Buhr, P. A., G. Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke (1991): "μC++: Concurrency in the object-oriented language C++." *Software — Practice and Experience*. Accepted for publication.

Gibbons, B. and D. Goldsmith (1991): "Proposal for compile time constants in class scope." Technical report, Apple Computer, Cupertino, CA, USA. ANSI document number X3J16/91-0067.

Insinga, A. (1991): "Working paper for a C++ string package." Technical report, Digital Equipment, Corp. ANSI document number X3J16/91-0078.

Lenkov, D. (1991): "Type identification in C++." Technical report, California Language Lab, Hewlett-Packard. ANSI document number X3J16/91-0063.

Lenkov, D., M. Mehta, and S. Unni (1991): "Type identification in C++." In *Proc. USENIX C++ Conference*, Washington, DC, USA. USENIX Association. April 1991.

O'RIORDAN, M. (1991a): "Namespace pollution." Technical report, Microsoft Corporation, Redmond, WA, USA. ANSI document number X3J16/91-0050.

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

ROWE, K. (1991): "Name space pollution: A proposal." Technical report, Microsoft Corp., Redmond, WA, USA. ANSI document number X3J16/91-0041.

SELIGER, R. (1990): "Extending C++ to support remote procedure call, concurrency, exception handling, and garbage collection." In *Proc. USENIX C++ Conference*, San Francisco, CA, USA. USENIX Association. April 9–11, 1990.

TURNER, JR., P. K. (1991): "Name lookup algorithms." Technical report, Language Processors, Inc. ANSI document number X3J16/91-0049.