ANSI C++ Committee Meeting, July 9-13, 1990

Brück, Dag M.

1990

CODEN: LUTFD2/(TFRT-7459)/1-17/(1990)

ANSI C++ Committee Meeting July 9–13, 1990

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
August 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
August 1990

Document Number
CODEN: LUTFD2/(TFRT-7459)/1-17/(1990)

Author(s)
Dag M. Brück

Supervisor

Sponsoring organisation
ABB Automation AB
Ericsson
Televerket

Title and subtitle
ANSI C++ Committee Meeting — July 9–13, 1990

Abstract

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

- A list of possible extension proposals; issues that are likely to be propsed during the lifetime of X3J16.
- Exception handling, in particular a discussion of termination vs. resumption semantics.
- Examples of how resumption can be abused.
- Real-time class libraries.

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
17

Security classification

Recipient's notes

1. Summary

The extensions group has been the most active working group of X3J16. The proposal for templates (also called parameterized types) was accepted for inclusion in the standard working document at the July meeting [Ellis and Stroustrup, 1990]. A proposal for overriding of inherited class names was discussed, and will be voted on at the November meeting. The main issue regarding extensions was exception handling, which is discussed separately below.

The library working group will concentrate on a few standard libraries, in particular strings and i/o-streams. Other libraries may be considered in the future, possibly including coroutines. The definition of standard libraries will of course depend strongly on key language extensions, such as, exception handling and templates.

The international working group is also quite active. The main tasks are to solicit international participation in X3J16, and to prepare the cooperation with the ISO C++ committee. X3J16 plans to hold one meeting (of three) every year outside USA, mainly as a sign of good will. Since the last meeting, Philippe Gautron, France, has joined the committee; total number of non-americans: 2. The main technical issue is currently national character sets [Simonsen and Stroustrup, 1990].

Jonathan Shopiro (AT&T Bell Laboratories) has replaced Margret Quinn as project editor. One advantage is that Shopiro has considerable experience of real-time systems. He is also interested in teaching a C++ course at the Department of Automatic Control in June 1991.

2. Language extensions

The following list of improvements that are likely to be proposed was written by Bjarne Stroustrup, and submitted to the members of the extension working group of X3J16. Reproduced with permission. [Comments by Dag Brück in brackets.]

Here is a first draft of the list of suggested "improvements." I mark things "urgent" if I think a decision is needed this year and "inevitable" if I think that C++ (or its successor) definitely will have the feature in 10 years. I don't make finer distinctions. Nor do I think we should start "solving" these problems. In fact, if we did that we would most likely dilute our efforts and distract the committee as a whole to the point where both extensions and the standardization of the core language would be endangered. I am posting the list as a start of classification and as a suggestion for common terminology.

Parameterized types: urgent. *Suggested base for discussion: My 1988 USENIX paper [Stroustrup, 1988] + ARM [Ellis and Stroustrup, 1990]. At the suggestion of Dmitry Lenkov, I have invited Sam Haradhvala from Object Design Inc. to present his experiences with an implementation of that scheme at the Seattle meeting. [Very good presentation and solid user experience. The proposal was included in the standard working document.] Difficulty: design almost done, implementation 3 to 6 months.*

Exception handling: urgent. *Suggested base for discussion: The papers Andy Koenig and I distributed at the NJ meeting plus the implementations and notes distributed at that meeting plus any further EH/C++ experience* *we can lay our hands on. I think it would be nice if we could present a definite proposal to the Silicon Valley meeting [November 1990]. Difficulty: design almost done, implementation 3 to 6 months. [See next section for further discussion.]*

Overriding: inevitable.

```
class cowboy { virtual draw(); };
class picture { virtual draw(); };
class animated_cowboy : public picture, public cowboy {
  // how do I overload and call the two distinct draw()s ?
};
```

Suggested base for discussion: The renaming discussion in the ARM. Difficulty: design simple, implementation a week. [The general idea was discussed and accepted. This feature will probably be included in the working document at the next meeting. For a detailed discussion, see Stroustrup, 1990; O'Riordan, 1990]

Indirect classes: inevitable. *In C++, objects of any class can be static, auto, and members of other classes. In each case the object is really allocated where it appears to be allocated. Other languages allocate all class objects on the free store and access them through handles only. This reduces the amount of recompilation caused by changes in object layout by insulating users. The cost is allocation, deallocation, and access overhead and C, Fortran, etc. object layout compatibility problems.*

I expect that mechanisms for defining C++ classes or objects so that they are implicitly placed in the free store but accessed by name rather than explicit pointer will be proposed. I expect the key mechanism in such schemes will be a keyword for saying 'objects of this class will be indirect' or 'this object will be indirect.' Without a 'lever' to distinguish indirect from ordinary classes we would encounter efficiency problems and severe compatibility problems.

Suggested base for discussion: We don't have one. Difficulty: design unknown, implementation unknown.

Meta classes: inevitable. *The more ambitious C++ systems are growing warts to support*

1. *Dynamic linking*
2. *Object I/O*
3. *Debugging using inspection of objects*
4. *Persistence*
5. *Calls across name spaces*

Eventually, someone must design a single interface to data structures supporting such activities. Otherwise we will all be re-inventing this wheel.

Suggested base for discussion: We don't have one. Difficulty: design hard, implementation 2 months.

I consider the design "hard" because there are a multitude of uses and the most general scheme involves a complete run-time representation of types and access control information. An important aspect of a design will be to avoid a meta-class facility to becomming either a way of subverting the type system or a major constraint of the C++ implementation techniques. Once designed, I don't see any major implementation problems. The compilers already have *all the information needed and can place it in memory when we have a design for run-time access to such information.*

Virtual data. *Whatever that may mean (I have seen several variants).*

Multi-methods. *CLOS-like function call based on two or more objects.*

Per-object protection. *In addition to the current per-class protection.*

Method combination. *Ala CLOS, in particular :before and :after methods.*

Delegation. *My attempt to define delegation for C++ failed. The issue will re-surface. It is somewhat related to indirect classes.*

Concurrency. *I expect we will see both minimal schemes (like Dag Brück's proposal) and maximal schemes (roughly along the lines of Ada's tasks).*

Persistence. *I expect we will see schemes specifically for persistence in addition to meta-class schemes supporting object I/O.*

Garbage collection. *I expect we will see two kinds of schemes: The conservative ones that do not affect the semantics of the language (except maybe for the calling of destructors) and the radical ones that require specific language changes that makes it impractical to run C++ programs without a garbage collector or a very large memory. [Edelson, 1990]*

3. Overriding

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 question is, how do I redefine and call the two inherited `draw()` functions? 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 currently not possible, and overriding was designed to preserve virtualness.

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

picture* p = new animated_cowboy;
cowboy*  c = new animated_cowboy;

p->draw();          // calls animated_cowboy::paint()
c->draw();          // calls animated_cowboy::shoot()
```

The manual text and the rationale for overriding will be presented at the November 1990 meeting.

4. Exception handling

Exception handling was the issue that raised the most controversy at the Seattle meeting. While the syntax proposed by Koenig and Stroustrup has been universally accepted, the choice between termination vs. resumption semantics is hotly debated. The different views have their origins in two mental models of what exception handling is all about:

- "get out of here" favours termination;
- "get help from somewhere" favours resumption.

Termination implies that all blocks between the 'throw' point and the handler are terminated. After the handler has been executed, the program continues with the statement following the handler. This model has already been used in Ada, and is probably well understood.

Resumption allows the program to continue execution with the statement following the 'throw' point, although the handler is located somewhere along the call chain. In this case, the stack cannot be discarded when the exception occurs, and the implementation is more complicated.

The lack of practical experience of exception handling makes this issue hard to discuss. Some people that have used resumption are now strongly against (e.g., Jim Mitchell, the implementor of Mesa), whereas users of other systems believe resumption is necessary in highly interactive systems (e.g., Microsoft and OS/2).

My view is that the resumption model is difficult to understand and implement, and that it will be very error-prone in practical use. Furthermore, most uses quoted for the need for resumption are variations of asynchronous interrupts, an area I think is more related to concurrency than exceptions. The protection of proprietary information has unfortunately prevented me from obtaining a detailed description of exceptions in OS/2, and how they migrate to the C++ environment developed at Microsoft.

The fundamental question raised is "what should be included in a standard?" The answer may range from a minimal set of features, essentially standardizing the current language, to a language with all sorts of potentially useful features. One of the main reasons for standardizing resumption is that there are "weak" arguments for not including resumption; people who do not want to use resumption do not have to.

A completely different area of potential problems was also discussed. Throwing an exception is similar to calling a function in the sense that a parameter object is passed. The same parameter matching rules that are used for function overloading should also be used for matching the handler; the problem is that handler matching is done at run-time which makes type checking more difficult. A type-secure system with a minimum of overhead would force the following restrictions on what can be thrown as an exception. The object cannot have:

- Private or protected base classes;
- Multiple non-virtual base classes;
- A private copy constructor;
- A private destructor.

I believe I can accept all of these restrictions, except the first one which I think violates current C++ idiom. A private base class is really an implementation detail; it is no more visible than a private member of a class. If an object with a private base class could not be used as an exception, this implementation detail would drasticly affect the "interface" of the derived class.

If private base classes were allowed, what should happen if we find a handler for the base class before we find a handler for the derived class? In my view, the private base class is "invisible," so no match would occur on the handler for the base class. There are other interesting issues as well; for example, what should be done if the copy constructor called to pass the exception raises an exception itself, or if the constructor for the exception object raises an exception?

The following examples show "abuse" or "creative use" of the resumption model for exception handling. They are in any case likely to be used effectively in C++.

Iteration

The following example is originally by Andrew Koenig. The exception handling mechanism is used to iterate over a vector of integers (note syntax for parameterized vector type). The iterator will throw an exception for every element; actual processing is made in the matching handler. The iterator makes a normal return when all elements have been processed.

```
int sum(Vector<int> v)
{
  int accum = 0;
  try {
    iterate(v);
  } catch (int i) {
    accum += i;
    continue;
  }
  return accum;
}

void iterate(Vector<int> v)
{
  for (int i = 0; i < v.n; i++)
    throw(v[i]);
}
```

If C++ would support nested function, like most useful languages, this example could be written as follows:

```
int sum(Vector<int> v)
{
  int accum = 0;
  void s(int i) { accum += i; }

  iterate(v, s);
  return accum;
}

typedef void (*PF)(int);
void iterate(Vector<int> v, PF func)
{
  for (int i = 0; i < v.n; i++)
    (*func)(v[i]);
}
```

The introduction of nested functions is a useful feature, although it represents a major change to the plain C model of computation. Exceptions with resumption probably requires the same run-time support as nested functions, but the advantages are not fully exploited.

The C++ Retriever

This example was invented by Dag Brück (after two beers). In this case, resumption implements Lisp-style dynamic binding of variables. Bjarne Stroustrup said: "this gives you Lisp functionality with Lisp efficiency and C++ syntactic ellegance."

Function g() asks for a variable binding in some stack frame by throwing a pointer. The handler in f() binds the pointer to the local variable x (note reference parameter), and resumes execution.

```
void f()
{
  int x = 2;
  try {
    g();
  } catch (int*& p) {
    p = &x;
    continue;
  }
}

void g()
{
  int* p = 0;
  throw(p);
  *p = 3;           // Assign to 'x' in f().
}
```

Any number of intermediate functions can be called between f() and g().

Coroutines

The following program is even more non-obvious than it seems:

```
void f()
{
  try {
    g();
  } catch (e2) {      // This is the handler we want, or...
  }
}

void g()
{
  try {
    h();
```

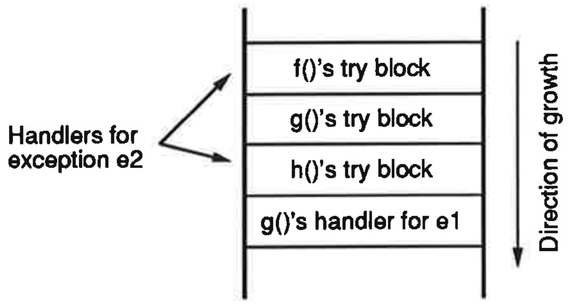

Figure 1. Typical stack frames when executing resumable handler.

```
  } catch (e1) {
    throw(e2);
  }
}

void h()
{
  try {
    throw(e1);        // This stack frame is still around,
  } catch (e2) {      // but is it visible?
  }
}
```

It is unclear what is the interpretation of this program. Figure 1 shows a typical stack layout when executing the handler in `g()`. There are two possible interpretations of what should happen when the handler in `g()` throws `e2`:

1. A handler for `e2` is located on the stack. In this case the closest matching handler is found in `h()`.
2. Function `g()` was called by `f()`, so the handler is located in `f()`. In this case some mechanism is needed to by-pass certain parts of the stack, or the runtime system must use a cactus stack for exception handling.

If the first interpretation (based on the actual layout of frames) is chosen, we have in fact implemented coroutines with the exception handling mechanism.

5. Concurrency

The Department of Automatic Control has established an informal cooperation with Microtec Research Inc. and The Software Components Group; both companies are located in Santa Clara, CA. The purpose is to develop a real-time kernel in C++, to be used on VME-bus systems. Automatic Control will develop the class library, and MRI and SCG will provide development systems and support. The class library will have the following features:

- Concurrent processes
- Semaphores, events
- Monitors
- Message passing
- Ada-like rendez-vous (?)
- Multiple scheduling algorithms (?)

- Signal processor classes (?)

Items marked (?) are likely to cause trouble, and will only be implemented as time permits. The class library will be used to control laboratory processes and an industrial robot. The project will be evaluated after one year.

There is also an increasing interest in X3J16 for concurrency, primarily in the form of coroutines. Several people have expressed interest in our work with MRI and SCG, and the library working group will consider a proposal for a coroutine class library, possibly to be included as one of the C++ standard libraries.

It is hopefully possible to design a coroutine class library that makes very few assumptions about its environment. This library could be used either for coroutine programming in a sequential system, or as the basis for a "hard" real-time system (which would require a more sophisticated implementation).

6. References

EDELSON, DANIEL R. (1990): "Dynamic Storage Reclamation in C++," UCSC-CRL-90-19, Board of Studies in Computer and Information Sciences, University of California at Santa Cruz, Santa Cruz, CA 95064.

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

HARADHVALA, SAM (1990): "Implementation of Parametrized Types," Object Design Inc., Burlington, MA, Slides presented 10 July 1990 at X3J16.

O'RIORDAN, MARTIN (1990): "Position Paper on Renaming for C++," Microsoft Corporation, Redmond, WA, USA.

SIMONSEN, KELD and BJARNE STROUSTRUP (1990): "A European Representation for ISO C,".

STROUSTRUP, BJARNE (1988): "Parametrized Types for C++," *Proceedings USENIX C++ Conference*, Denver, Colorado.

STROUSTRUP, BJARNE (1990): "A Proposal for Generalization of Overriding," AT&T Bell Laboratories, Murray Hill, NJ 07974, ANSI document number X3J16/90-0046.

Appendix

This appendix contains copies of the slides Sam Haradhvala used for presenting Object Design's implementation of templates (parameterized types). This implementation will probably be part of the next AT&T compiler.

Implementation of PT

Sam Haradhvala

Object Design, Inc.

1 New England Executiv Park

Burlington, MA 01803

USA

sam@odi.com

Why PT?

Initially, to support ObjectStore aggregates.

Also turned out to be a really useful language feature that was of general utility

Can't live without them, they are permeating all new code

The PT Language Subset

The implementation supports the subset of the PT facility as it relates to the parameterization of classes.

The following features are supported by the implementation:

Class templates

Member Function templates

Specific Member Functions to override default member function templates

Terminology

Class template:

```
template <class T> class foo :
```

foo is the class template

Member function template:

```
template <class T> void foo<T>::bar()
                   {...}
```

Template class

```
foo<int> is a template class.
```

Example: Class Template

```
emplate <class T> class Stack{
  int  size;
  int  ptr;
  T**   s;

public:
  Stack(int sz);
  ~Stack();

  void push(T& item);
  T& pop();
  T& top();

  int empty() const
    {return ptr == 0;}
  int full() const
    {return ptr == size;}
  void error(const char* msg)
    { fprintf (stderr,"%s",msg);
      exit(-1) ;}
};;
```

Member Function Template

```
template <class T>
Stack<T>::Stack(int sz)
{ s = new T* [size = sz] ;
  ptr = 0;
}

template <class T> Stack<T>::~Stack()
{ delete [size] s;}

template <class T>
void Stack<T>::push(T& item)
{ if (full())
    error("Stack overflow");
  s[ptr++] = & item;
}

template <class T>
T &Stack<T>::pop()
{ if (empty())
    error("Stack Underflow.");
  return *s[--ptr];
}
```

Example:Specific Member Function

```
// Specific pop for Stack<int>::pop()
int &Stack<int>::pop()
{

}
```

Implementation Overview

The implementation is based upon AT&T 2.0 cfront.

The PT implementation tries to be as non-intrusive as possible. It mainly deals with the mechanics of maintaining templates, and generating template classes and bodies in terms of the Intermediate Language used by cfront, which then carries out the usual analysis and code generation from the IL.

The general strategy should be adaptable to any C++ front end that has a clean interface between the syntax and semantic analysis phases of compilation.

Compiling PT (salient points)

There are a couple of noteworthy differences when compiling parameterized types:

The syntax analysis of the class template and it's member function templates must be done in a scope that includes unbound "type type" arguments.

The semantic analysis of the template class or member may not immediately follow the syntax analysis for the template definition, since in general the use of the template class is likely to be separated by intervening text.

The environment for the semantic analysis, must be the environment that existed at the point of definition of the class or member function template.

Syntax Analysis

Class and function member template definitions are parsed and retained as syntax trees when encountered in the source. The template arguments shadow any globals with the same name during this phase.

"type type" formals are treated as typedef names during the course of the parse.

Expression formals are treated as unbound variable names

The result of the analysis is a syntax tree that is associated with the template definition.

Semantic Analysis of Template Classes

Semantic analysis is initiated whenever a new template class is encountered during the course of the compilation. It involves the following steps:

Generate a copy of the saved syntax tree.

Bind the formal template arguments.

*Establish the name environment as it existed at the time of the template definition.

Perform the usual semantic processing on the class.

14

Cascading Semantic Analysis

It is possible that before semantic analysis can be carried out on a template class, or template function, the template classes that it in turn depends upon must be analyzed.

```
template <class T> class c1 {T v;} ;

template <class T> class c2: c1<T>{}:

c2<int> ;
```

Thus, although c2<int> provokes the semantic analysis, c1<int> must be analyzed before c2<int> can be.

Semantic Analysis of Template Member Functions

Semantic analysis of template member functions involves the same processing steps as those used for template classes.

The hard part is deciding when and where to generate these definitions.

In our approach, the compiler always generates the function definition, if the function template was encountered during the compilation.

Generation of function definitions can be excluded via a pragma.

Compilation Model

The compilation model follows the usual C++ mode of compilation, with the source files containing parameterized definitions being maintained exactly like any other C++ source file.

Define the class template in ".h" files that are included as usual.

Define the member function templates in ".c" files that suitable for inclusion.

The ".h" files are included whenever access is required to a class template.

The ".c" file is only included in a specific file that names all the template classes that need their member functions defined.

Pros and Cons

It is simple, it fits well into the usual ways of maintaining and compiling C++ source.

It makes no special requirements of the development environment, or require any additional tools.

It places responsibility for defining member functions directly on the user.

Error messages resulting from the use of "incorrect" "type type" parameters is delayed until the body definition is encountered.

More sophisticated schemes are easy to envision, perhaps not as easy to implement.

Example: Generated C code

```
typedef int TStack__pt__2_i :
struct Stack__pt__2_i {* sizeof Stack__pt__2_i -- 12 */
int size__15Stack__pt__2_i :
int ptr__15Stack__pt__2_i :
TStack__pt__2_i **s__15Stack__pt__2_i :
}:
struct Stack__pt__2_i *__ct__15Stack__pt__2_iFi (__0this . __0sz
)
register struct Stack__pt__2_i *__0this :
int __0sz :
{ if (__0this || (__0this - (struct Stack__pt__2_i *)__nw__FUi (
sizeof (struct Stack__pt__2_i)) )){
__0this -> s__15Stack__pt__2_i - (int **)(((int **)__nw__FUi
((sizeof (TStack__pt__2_i *))* (__0this -> size__15Stack__pt__2_i
- __0sz )) )):
__0this -> ptr__15Stack__pt__2_i - 0 :
}
return __0this :}
```

P.S. fortunately our debugger does understand how to unmangle parameterized types

Example: File organization for use of PT stack

Stack.h defines the class template "Stack"

Stack.c defines the member function templates for Stack

Regular C++ .c files simply include "Stack.h"

```
#include "Stack.h"
...
stack<int> stack_of_int ;
stack<double> stack_of_double ;
...
```

Designate a distinct file to hold the bodies

```
#include "Stack.h"
#include "Stack.c"
...
typedef stack<int> si ;
typedef stack<double> sd ;
...
```

17

Implementation Statistics

Template processing: ~3100 lines

Tree copying utilities: ~1700 lines

Misc. cfront modifications (grammar, lexer, etc.): ~500 lines

Tools: filt, debugger etc. ~ 500 lines