ANSI Doc No:	X3J16/93-0014
ISO Doc No:	WG21/N0222
Date:	December 10, 1992
Project:	Prog. Language C++
Reply To:	Dag M. Brück
	dag@control.lth.se

Case-study: Casting in C++

1. Introduction

Jerry Schwarz' paper on `~const` [Schwarz, 1993] made me look a little more carefully at the software system for modelling and simulation of control systems that we're developing at my department. Here's first some figures so you can estimate the size of our system. I have counted the number of lines with a `;` or `{`, which gives a reasonable approximation of the number of declarations and statements:

Type of code	Count	Total
Header files	4 199	
Impl. user interface	3 536	
Impl. representation kernel	13 314	21 049
InterViews Graphical UI	19 632	
FORTRAN numerical code	3 983	
C++ Matrix library	4 204	
Other libraries	2 202	30 021

This survey relates to the 21 049 "lines" of C++ code that we have written ourselves. Judy Grass at AT&T Bell Laboratories looked at an earlier version with CIA++, and in her view this code is unusually complex (it broke both CIA++ and the database interface).

2. Results

I have studied every cast in our code and tried to classify them as static cast, dynamic cast, re-interpret cast and const cast. The results are:

Type of cast	# casts	# questionable
Const cast, direct application of `~const`	9	1
Other const cast	21	4
Dynamic cast	212	1
Static cast	24	7
Re-interpret cast	15	0
Cast to same type	2	2
TOTAL	283	15

Questionable casts can be eliminated by trivial modifications of the code, so we shouldn't really count them.

3. Discussion

Dynamic cast

We have four separate class hierarchies that use dynamic casts extensively, much more often than I anticipated. This is of course a reason for concern, but I haven't (yet) found any obvious design error in the class hierarchies.

Much of our code is structured as functions that operate on a pair of class objects, instead of a member function that takes one class object. We think this is a natural syntax for symmetric operations.

In many of these cases I think we could have re-written the code to use many more virtual functions, but the drawback would be that all classes except one in the class hierarchy would have empty realizations. I don't like that either. For example, in a class hierarchy for representing expression trees, only the class for function calls needs a member function `GetParameters()`. Should we pollute classes for variables and literals with that function?

Const cast

The number of const casts is surprisingly small. A closer study shows that the people in my department have listened carefully to my declaration that "casts are unsafe, don't use them." People have written code that obeys the rule of bitwise constness instead of meaningwise constness. Instead of casting away const, member functions that are logically const have been declared non-const because they perform some changes behind the scenes. This effect propagates all over the system because a const member function cannot call a non-const member function without casting away const. For the same reason, arguments that are logically constant have been declared non-const because we call a non-const member function of the argument.

We have implemented dynamic casting by downcasting functions in the base class, e.g., `Derived* asDerived()`. This approach works quite well because there is reasonable number of classes and we have complete control over that part of the code. However, we suffer from a major design error on my part: we haven't overloaded the downcasting functions on const. For every existing downcasting function there should also exist `const Derived* asDerived() const`. This may be a major reason for the problems we have with missing constness, but I haven't had a chance to asess the impact on our code. Automatic overloading on constness may be an advantage of downcasting functions over `dynamic_cast<T>` in the RTTI proposal.

The matrix library we use generated a large number (over one hundred) of warnings because it did not support meaningwise constness; a typical message was "temporary bound to non-const reference." A later version, NEWMAT06, has been cleaned up and all our code now uses it without warnings.

Static cast

All static casts may in a broad sense be regarded as questionable. There are three main uses of static casts in our application:

- There is an awkward conversion between two similar types `String` and `Name`, which could be avoided with additional overloading.

- We have to shut the compiler up when we delete something through a reference (`delete &r`).
- A few static casts are forced on us by mistakes in InterViews. A public virtual function in a base class is declared as protected in a derived class.

We should probably introduce a few additional static casts for converting floating point numbers to integers, just to highlight that this is a cast where information is lost.

Re-interpret cast

All re-interpret casts have a particular use. We call numerical routines in FORTRAN that take an integer parameter vector; the parameter vector is passed to our residual routine, written in C++. We use one of the parameters as a pointer to an integration status object, which of course involves a cast.

4. Conclusions

In any case, I learned three things from this study:

1. Get your facts straight first, argue later.
2. The problem of meaningwise constness is not easily solved. It fundamentally affects the way we program, and it propagates through-out the code.
3. The case for `~const` is not as clear-cut as I initially thought. I expected to be able to remove n const-casts, but instead I would have to make a bigger redesign to support meaningwise constness uniformly.

I believe in meaningwise constness. Our software has simply drifted towards bitwise constness without any real discussion of the design in our group. The reasons are my overly simplified statement about casts, combined with hard error messages from the compiler. A clean-up at this stage would be a major effort, but possibly worth the trouble.

5. References

SCHWARZ, J. S. (1993): "A minimal `~const` proposal." Technical Report, Lucid, Inc. Document number X3J16/93-0005 and WG21/N217.