Accredited Standards Committee
X3, INFORMATION PROCESSING SYSTEMS*

Doc No: X3J16/92-0134
WG21/N0210
Date: 18 November 1992
Project: 738-D (PL C++)
Reply To: Stephen D. Clamage

Why Translation Limits?

Stephen D. Clamage
TauMetric Corporation
steve@taumet.com

Introduction

The subject of translation limit specifications in the C++ Standard has been unsettled for some time. The subject was discussed in full committee in Dallas in November 1991 (see *Minutes*, X3J16/91-0136, WG21/N0069, pages 16-18). It was unexpectedly brought up for a vote in Boston in November 1992. There is an assumption on the part of many Committee members that the final Standard will contain a specification of translation limits for conforming C++ implementations. Not everyone agrees on what form these limits should take, nor on whether the Standard should address limits at all. In this paper I summarize my understanding of the issues and make specific recommendations.

Background

The C Standard includes a section on translation limits, providing statements of minimum capacity which any conforming C implementation must satisfy. The C Standard requires that an implementation "be able to translate and execute at least one program that contains at least one instance of every one of the ... limits". In practice, this means that a vendor may supply one such program as an example and meet this requirement. This has been called a "rubber teeth" requirement, since many of the limits are quite modest, and user programs which fall within the limits are still allowed to fail.

At the C++ Committee meeting in Dallas in November 1991 it was suggested that two sets of limits be provided. One set would have rather high limits, and C++ implementations would have to handle programs which hit no more than one of these limits. A second set would resemble the limits in the C Standard, with the requirement that the implementation would have to handle one program which hit all these limits. A proposal along these lines was presented for a vote in Boston in November 1992, but was rejected at least in part due to insufficient preparation.

Why should we specify limits?

The following is a summary of reasons which have been advanced for specifying limits in the Standard (listed in arbitrary order).

---

* *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 1250 Eye St. NW, Suite 200, Washington, DC 20005

1. The C Standard has them, so the C++ Standard should have them too.

2. Reasonable limits would ensure that "toy" implementations could not claim to be standard-conforming.

3. Programmers and program managers want known size and complexity limits within which they are ensured their programs are portable.

4. Those who specify or sell validation suites need guidelines for creating tests. This is related to #3, and lets a validation suite be created which will be valid across many implementations.

Rebuttal to these arguments

Here are brief counter-arguments, expanded in later sections.

1. The C Standard specifies limits, but no other language standard does; the point is relevant only in the context of C compatibility. I argue below that it isn't really relevant.

2. Do we care whether a toy implementation can claim to be standard-conforming? If it is not useful, no one will want it. If it is useful, why should we disparage it?

3. It has been observed that "minima become maxima". If the Standard requires only N of something in a program or module, then programmers in many environments will be forbidden to use N+1 of them. Vendors will point to the Standard as an excuse to reject programs containing N+1. This is perhaps a theoretical rebuttal, but see more discussion below.

4. An alternative to predefined limits for test suites is to specify or create a test suite which requires certain capabilities, and publicize them. C++ implementations which do not have the capacity will not pass the tests. Users with smaller requirements will look for smaller test suites which these implementations *can* pass.

Detailed discussion

It seems that we would like to say that standard-conforming implementations must have a reasonable capacity. To this end, we would set some minimum limits and say that implementations must be able to translate and execute any conforming program which does not exceed the limits.

Unfortunately, we can't say this. There is no way to test, or even to write, all conforming programs which fall within the limits. It is bad practice for a standard to require something which cannot be tested or otherwise verified. This leads to "rubber teeth" requirements of dubious usefulness.

How do we go about picking a limit for the Standard? Let's consider the distinguished characters in an external identifier as a concrete example.

The C Standard requires a minimum of six characters, and upper and lower case need not be distinguished. This very modest limit was set because some widely used computer systems had such limits. By the time the C Standard was finally adopted, those widely-used systems were not so widely used any more, and the limit began to look rather silly. On the other hand, if the proposed standard had specified more characters or mixed case, it could not have been adopted; there would have been too much opposition from system suppliers and users.

Today, some widely-used systems have a limit of 30 to 32 characters, and at least one of them supports only upper case external identifiers. Should we take 30 as the limit? It really isn't enough to support "name mangling" as commonly used. A *much* larger limit (at least two orders of magnitude larger) would be very desirable. There would likely be opposition to such a large limit from suppliers and users of some systems — it would be impossible or inconvenient to have a conforming implementation on those systems. I don't know how to resolve this, and I suspect that there is no good solution. Any number you pick will offend someone.

Let's look at limits from another perspective. Suppose that the Standard says that N *fizzbins* must be supported. (A *fizzbin* is just something for which we set a limit.)

What if the limit is set low? If an application absolutely requires many more than N fizzbins, the assurance that N are available is of no interest (like 6-character external names). Compiler vendors might feel encouraged to build in a limit of N fizzbins, since they can't be held to account for more than N. The limit in the Standard in this case proves harmful rather than helpful.

On the other hand, perhaps it will become generally recognized that N fizzbins is unrealistically low, and all C++ implementations will make more than N available (like characters in external names). The limit in the Standard in this case is useless, since programmers will expect more than N, but will not know how many will be available.

What if the limit is set high? Suppose that a computer system of interest cannot possibly support N fizzbins (like 64K external variables on a system with only 64K bytes of memory). It is now impossible to have a conforming implementation for this system, or perhaps possible only with significant performance penalties. This is another case where the limit in the Standard proves harmful. If you are programming a microwave oven, you don't need 64,000 external variables, or 2000 virtual functions in a class, but you would still like to have a standard-conforming compiler.

It seems that we must emulate Baby Bear and pick limits which are not too large, not too small, but "Just Right." What limits will be "Just Right" for programming the telephone system for a continent and also for programming microwave ovens? Once again, I believe this is an insoluble problem.

It has been suggested that these limits be made the subject of a separate category of conformance. The Standard would specify "language conformance" and "environment conformance", for example. This leads to a possibly open-ended set of levels or classes of standards conformance, a concept which has precedent, but which was rejected by the Committee.

Absence of limits

Let's now consider the situation if the C++ Standard mentions no translation limits at all.

First of all, no implementation could appeal to the Standard in rejecting a large or complex program. This is precisely the case with other programming languages. Compiler capacity becomes strictly a "quality of implementation" issue, along with many other such issues.

Anyone purchasing a C++ implementation has many things to consider. If a large application is to be developed, the implementation must support developing a large application, no matter what the Standard says on the subject. If an alarm clock is being programmed, the capacity of the compiler will not be an issue.

Suppliers of test suites will have to resolve for themselves and their clients how complex their tests will be. They will have to pick their own limits. Maybe they will want to pick different sets of limits and have different sets of tests. The test suite supplier will publish the minimum requirements for a system to be tested, just as application programs commonly specify processor, memory and disk requirements. Test suite users can decide whether the published limits are appropriate to their needs and pick a test suite accordingly. In the end, either the C++ Committee picks limits which everyone else has to live with for many years, or suppliers and their clients negotiate jointly according to their needs.

Closing comments

Modern compilers tend not to have any fixed limits other than those imposed by the system on which they are running. You won't usually find a limit on just nesting of braces, or on just the number of members of a class, but rather a limit on the total system resources. Once those resources (real memory, virtual memory, disk blocks, CPU minutes) are exhausted, it will do no good to require that the C++ implementation handle more.

Now suppose that an implementation is created which does have fixed limits, but gets a terrific speed advantage. Potential customers would have the option of picking a very fast system with capacity limits. If they find this valuable, why should the limits prevent it from being standard-conforming? If the implementation is so limited that no useful programs can be written, why is this a standards issue? No one will want it, so it won't matter whether it claims to be standard-conforming.

There are many "quality of implementation" issues on which the C and C++ Standards are and must be silent, because there is nothing portable which can be said. For example, a C implementation which requires 100 Mb of real memory and 100 hours of computer time to compile and run the null program

```
main() { return 0; }
```

can be standard-conforming. Yet an implementation which allows only 126 local variables in a block is *not* standard-conforming, no matter how wonderful its other qualities. This seems strange to me.

I propose that we put translation limits strictly in the "quality of implementation" category, and impose no limits.