11 February 1991

Mr. Bjarne Stroustrup
AT&T Bell Laboratories
Murray Hill, NJ
Tel: (201) 582-7393

Dear Mr. Stroustrup,

I enclose a (corrected copy) of a letter I sent recently to the X3J16 ANSI C++ Standards committee. Perhaps you have already considered these issues; if not, I hope you find my comments useful and perhaps even thought provoking.

Cordially,

Craig Franklin
Vice-President
Marketing

510 Castillo Street, Santa Barbara, California 93101 • (805) 965-6044 • FAX (805) 965-6343

1 February 1991

X3J16
c/o Dmitry Lenkov
Hewlett-Packard California Language Lab
Mail Stop: 47LE
19447 Pruneridge Avenue
Cupertino, CA 95014
Tel: (408) 447-5279
Fax: (408) 447-4924

Dear X3J16:

Although I am a C++ novice, I think my learning experiences with C++ could be valuable to X3J16. I spent 13 years on X3J1, the PL/I Standards Committee, so I also understand something about the procedures followed by standards committees.

Please consider what follows to be a formal input to X3J16, to be placed on the agenda for the first meeting for which it meets the appropriate deadline.

I hope the committee will consider my relative ignorance and inexperience with C++ to be an asset rather than a liability. During the past 30 years, I have programmed tens of thousands of lines of working code in 20 different languages, including Algol-60, Fortran, LISP, PL/I, Pascal, and C, but I am still a C++ novice. If some of the issues I raise below already have well-known answers, let me apologize in advance.

With that caveat, let us proceed.

There seem to be two different styles permitted when defining a new class. I don't know if these C++ styles have already been given names, so I will call them the Interface/Body style and the Integrated style. I prefer the Integrated style. It is usually shorter, since the interface definition and the actual implementation for each method are placed together in the text.

The "white rat" I will use for all my examples is a class, named `U64`, for performing unsigned 64 bit arithmetic. The goal of `class U64` is to be able to take any working C++ program which uses 32 bit arithmetic, include `U64` at the head of it, change certain selected declarations from `unsigned long` to `U64`, and to have the newly modified C++ program compile and execute correctly, with no other editing changes. This is a tough challenge. If C++ can meet it, then it would be fair to say that as an extensible language, it is in a certain sense *complete*.

Here is a small part of class U64 in Integrated style:

```
typedef     unsigned long       U32;

class U64   { U32 L; U32 H;                 // Exchange L and H for Big Endian

private:

friend      U64   UADD_64_64(U64,U64);      // U64 + U64 -> U64
friend      U64   USUB_64_64(U64,U64);      // U64 - U64 -> U64

public:

      U64()                                 { }
      U64(U32 l,U32 h=0)                    { L = l; H = h; }

friend      U64 operator - (U64 i)          { return U64(0) - i; }
friend      U64 operator ~ (U64 i)          { return U64(~i.L, ~i.H); }
friend      U32 operator ! (U64 i)          { return (i.L | i.H) == 0; }

friend      U64 operator + (U64 i,U64 j)    { return UADD_64_64(i,j); }
friend      U64 operator - (U64 i,U64 j)    { return USUB_64_64(i,j); }

            U64 operator += (U64 i)         { return *this = *this + i; }
            U64 operator -= (U64 i)         { return *this = *this - i; }

}; /* End U64 Class */

extern      U64   UADD_64_64(U64,U64);      // U64 + U64 -> U64 in assembler
extern      U64   USUB_64_64(U64,U64);      // U64 - U64 -> U64 in assembler
```

And here is the same small part of class U64 in Interface/Body style:

```
typedef     unsigned long      U32;

class U64   { U32 L; U32 H;                      // Exchange L and H for Big Endian

private:

friend      U64   UADD_64_64(U64,U64);           // U64 + U64 -> U64
friend      U64   USUB_64_64(U64,U64);           // U64 - U64 -> U64

public:

            U64()              { }
            U64(U32 l,U32 h=0) { L = l; H = h; }

friend      U64 operator - (U64 i);
friend      U64 operator ~ (U64 i);
friend      U32 operator ! (U64 i);

friend      U64 operator + (U64 i,U64 j);
friend      U64 operator - (U64 i,U64 j);

            U64 operator += (U64 i);
            U64 operator -= (U64 i);

}; /* End U64 Class */

extern      U64 UADD_64_64(U64,U64); // U64 + U64 -> U64
extern      U64 USUB_64_64(U64,U64); // U64 - U64 -> U64

inline      U64 operator - (U64 i)          { return U64(0) - i; }
inline      U64 operator ~ (U64 i)          { return U64(~i.L, ~i.H); }
inline      U32 operator ! (U64 i)          { return (i.L | i.H) == 0; }

inline      U64 operator + (U64 i,U64 j)    { return UADD_64_64(i,j); }
inline      U64 operator - (U64 i,U64 j)    { return USUB_64_64(i,j); }

inline      U64 operator += (U64 i)         { return *this = *this + i; }
inline      U64 operator -= (U64 i)         { return *this = *this - i; }
```

**Discussion**

There are several things about U64 that were certainly non-obvious to me when I programmed it:

1. In the Interface/Body style, I can make the Body of each method be inline or not, as I choose. In the Integrated style, friend seems to imply inline. Since I personally prefer the Integrated style, I have lost a capability here. I would like to be able to write:

```
inline friend      U64 operator ! (U64 i)      { return (i.L | i.H) == 0; }
```

or

```
noinline friend    U64 operator ! (U64 i)      { return (i.L | i.H) == 0; }
```

Therefore, as my first formal suggestion, I would like to see inline friend added to C++. This will restore to the Integrated style the same optional inline capability as for the Interface/Body style.

2. Now the question arises as to what should be the default, inline or noinline? As the sainted Dijkstra has remarked, additional information which the compiler can use to make better code should be hints (which the compiler can ignore if it wishes), and the default should be to omit the hint.

C and C++ follow exactly this philosophy with respect to the register keyword. It is a hint, the compiler can ignore it, the compiler can give a warning and delete register if the & operator is applied to the register variable, and a very smart compiler can even add an implicit register to a declaration if it can determine that it is safe to do so.

From this argument, the default should be noinline and you should have to write the inline optimization hint explicitly. Of course, this has the major disadvantage that existing practice is the other way: friend in this context implies inline. Since X3 committees are supposed to codify existing practice, this is a very good argument. I don't care how the committee resolves this debate. What I want is not to be penalized when I write in the Integrated style. I want to have the same power that a programmer writing in the Interface/Body style has. Right now, I don't.

(I defer any arguments in favor of my preferred style. It is not usually useful to argue about style. But since C++ permits both styles, it should give them both the same power. I hope the committee will not interpret this as an argument in favor of deleting the Integrated style.)

So my second formal request to the committee, under the assumption that inline friend above was added to C++, is to review, consider, and decide if inline should be the default, or should it be explicitly required? (If inline is the default, then the committee next needs to consider whether the noinline keyword should be added to C++ so the inline default can be overridden.)

3. When I convert a class from the Interface/Body style to the Integrated style, and the Body method is `inline`, I at least have a way to express this in the Integrated style. When the Body method is `extern`, however, I can't. That is, I am not permitted to say:

```
...
private:

extern friend U64 UADD_64_64(U64,U64);      // U64 + U64 -> U64 in assembler
extern friend U64 USUB_64_64(U64,U64);      // U64 - U64 -> U64 in assembler
...
```

Instead, I am forced to move the `extern` information to a different spatial location:

```
...
private:

friend U64 UADD_64_64(U64,U64);      // U64 + U64 -> U64
friend U64 USUB_64_64(U64,U64);      // U64 - U64 -> U64
...
...
...

extern U64 UADD_64_64(U64,U64);      // U64 + U64 -> U64 in assembler
extern U64 USUB_64_64(U64,U64);      // U64 - U64 -> U64 in assembler
```

In fact, the `extern` declarations are not even inside `class U64`.

Therefore, my third formal request to the committee is to permit `extern` with `friend`.

The arguments for `extern friend` are different than the arguments for allowing `inline` in the same position. In that case, `inline` was a hint. In this case, `extern` is a semantic necessity, and I do not want, for reasons of documentation, maintenance, clarity, and consistency, to separate it to a far removed textual location. In today's C++, I am in effect forced to write two identical prototypes, one to carry `friend` and one to carry `extern`. I suspect that what I have just said is not the approved C++ way of looking at the world, but that is how it looks to at least one novice, namely me. (It is true that if I change one prototype and forget to change the other one, the compiler may well complain about it. I do not regard this as a compelling argument for the current state of affairs.)

4. My fourth point is what I regard as an even more interesting lacunae. Suppose my program contains this fragment:

```
U64 i;
...
if (i) printf("i != 0");
```

But this is wrong. It must be rewritten as:

```
if (i != 0) printf("i != 0");
```

The reason is that the C++ translator looks at `if(i)` and tries to convert `i` to integer, by looking in `class U64` for an `operator int`.

But this is not what I want. I might have defined an `operator int`, which would return the

low order part of `i`, that is, it would truncate it. But for the purposes of the `if`, this is the wrong thing to do. The `if` wants to know if `i`, including the high order part, is all zero or not. (This problem does not arise in the classic complex case, because complex numbers are not ordered, and programs never ask `if (C)`, where `C` is complex.)

I would like to committee to consider formally these four possible suggestions for solving this problem:

4a. Add `operator if` to C++ and use it for `?` contexts also.

4b. Add `operator ?` to C++, and use it for `if` contexts also.

4c. Add both `operator if` and `operator ?` to C++.

4d. The C++ translator should convert `if(i)` to `if (i!=0)` first, and similarly for `?`. Then the class need only supply the `!=` operator.

I personally prefer choice 4c, combined with my default suggestion below.

If you leave matters the way they are, I have to find and edit every occurrence of `if(i)` when I change the declaration of `i` from one of the standard integer-like datatypes to `class U64`. This seems to me to violate "The Spirit Of C++," if there is such a thing.

6. I find it tedious to define `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`. I personally believe that the translator can and should supply the obvious meanings, if the corresponding binary operator is given. If the translator is not willing to do that by default, perhaps it will at least let me request it, by something like the following:

```
default: { +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= }.
```

My formal request is to have the committee consider this issue. I don't care about the syntax; I just want the functionality.

7. Similarly, we can surely agree that if unary `-` is not supplied, but binary `-` is, that the translator can, upon encountering `-i`, convert this to `(0-i)`, with the well-defined binary meaning.

In exactly similar fashion, if unary `~` is missing, then `~i` can and should be converted to `(-1^i)`

Finally, one could ask if the same logic applies to the transformation of `!i` to `(i==0)`.

If all three of these transformations are admitted, then we don't need to define any of the three unary operators `{-, ~, !}`, just as the previous item proposed eliminating the definitions of the reflexive binary operators. I regard all such simplifications, either automatic or requested, as making C++ more user friendly for the non-expert writer of new classes.

One possible syntax is:

```
default: { -, ~, ! }
```

My formal request is to have the committee consider this issue. Again, I don't care about the syntax; I just want the functionality.

Note that if proposal 4c is adopted, to add `operator if` and `operator ?`, then one could reasonably write

```
default: { if, ? }
```

which requests the effect of proposal 4d.

8. I know of no way to request the translator to tell me if there are any operators I have missed defining. But if there is one feature a programmer of a new class needs, it is to know when the job is done.

Any suggestions?

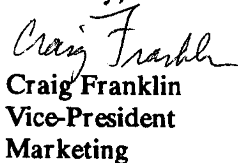
Summary

As you can see, I have directed my attention in this letter primarily to issues of ease of use of C++ by someone trying to write a new class. I fully realize that the majority of C++ users plan to re-use existing, debugged classes written by a C++ expert, and presumably the expert has been using C++ for many years without any of the user-friendly features I discuss above.

But if you really want C++ to be more widely used, don't you need to make it easier to write, debug, and document new classes, as well as just to use them?

To paraphrase a comment often heard on the ANSI PL/I committee, "The vast majority of C++ programs are yet to be written. That is why it is important in the committee to standardize it right." The first sentence of this comment turned out to be true for PL/I, but only just barely. I hope you have better success, and I hope that my input above is of some use to the committee.

Cordially,



Craig Franklin
Vice-President
Marketing