We put the PRO in programming!! (tm)

0. Abstract

C++ is becoming a growing celebrity among C programmers, object-oriented crowds, and the programming community at large. It has gained enough interest, acceptance, and marketing force that an organizational meeting to "ANSI"fy C++ was held on December 15, 1989 in Washington, DC. Bjarne Stroustrup, inventor and till now general "keeper" of C++, was inviting as a guest speaker. At that meeting, Bjarne provided some wise thoughts, comments, and pleas to the new committee. His comments were about the importance of proper standardization of C++. A slightly annotated transcript of his talk follows.

1. Salutations

Thank you for inviting me; it's a great honor. It's nice to have all the interest in C++ currently in progress and I'm pleased to see so many familiar faces here. I very much hope that the C++ community, as some of you are part of, will take part in this standardization process. We don't want to get a complete change of personnel as we go from the informal days to the more formal procedures of the standardization process. That could be quite devastating.

I will skip the importance of standardization and the advantages of standardization because if you didn't at least have some inclining of that you wouldn't be here. We also have very limited time for this talk. I will therefore focus more on the dangers of standardization. Not that I think that the dangers dominate, but because we need to face them to avoid them.

2. Standard Desires

[1] We need a standard tomorrow

We will not get that
It could take 5 years
It must not take 5 years

The reason we are here is that we want a standard .... and clearly we want it yesterday. Equally clear is that we are not going to get that.

I have worked for about 10 years on language design and C++ in particular. The last year to year and a half has been spent primarily on getting the definition more solid. The intent was to make a good foundation for standardization.

But, we're all here because we don't have a standard. Now that presents a real danger: we may work for 5 years and still not have a standard. Think about it: that IS a real danger.

We don't have to take 5 years. In some sense if we take 5 years, we've failed. So, what we have to look at is what we want - and how to get it in a reasonable time frame. This is very important.

21-24 120th Street • Richmond Hill, NY • 11418

3. Reference Manual Considerations

[2] We want a reference manual that is

precise
non-evasive
comprehensive
comprehensible

I was trying to figure out what we wanted and came up with a few considerations.

We want a manual that's precise. Clearly. Another thing we want in a reference manual is for it to be non-evasive. I hope that should be clear too. We don't want to get a standard that's a standard just because we agreed to leave things undefined and vague. That would be easier, however. We also want it be comprehensive. That is, it must possess two things. The standard must cover things in sufficient depth so that you actually can understand the language. It also needs to define enough of the language so that you can actually write programs in what is precisely defined. The point is that we don't just want to standardize the easy central part. This would force C++ programmers to use ill-defined extension and features.

Finally, we want a manual to be comprehensible. I emphasize that mainly in contrast to the most precise, least evasive, and the most comprehensive manual I've even seen: the Algol-68 manual. However, it choked the language by fulfilling the first three promises but not the fourth.

4. Tension in C++ Standardization

[3] Tensions:

perfection vs timeliness
perfection vs compatibility
users vs implementors
low-level vs high-level constructs
programming
    -in-the-small vs -in-the-large
compatibility
    within product line vs across product lines
(remember:
    there will always be someone with
    a genuine need for the opposite bias)

In attempting to reach our goals here, we're going to see some sort of semi-institutionalized tensions. The first one is that some people want the perfect language. That is to be expected, and in theory it is good. However, on the other hand, *those people are not going to get that on time*. If we go off chasing perfection we don't get a standard on time either. This is important to consider.

Similarly, perfection must be balanced against compatibility. I will speak of this later. For the time being, consider that we will be having big problems deciding what is compatible with what, with what our standard ought to be compatible with, and exactly what "compatible" means. This will be very stressful.

Perfection versus compatibility can become a tug-of-war over where the complexity should be put (in language features or in libraries), about when the work should be put in (at compile time or at run-time), and about who should do the work (implementors or users). Let's not get into such a state. On this same thought, let's not get into a situation where the users and the implementors of C++ collide head-on.

Consider low-level and high-level constructs. They are both important. According to L. Peter Deutch, C++ has the largest spread from the highest level features to the lowest level features. Think about what this says for a moment. This means that some people can live their whole "working life" without ever having to use something on the other end of the language's spectrum. Please consider this because the possibility exists that you can spend your whole working life with C++ and simply never meet anybody who lives on the other end of the spectrum. They do indeed exist, though, and perhaps in the same numbers as you and your friends. So, please, remember, that even if you haven't met them, there are people whose working lives do indeed depend upon the features from the other end of the spectrum that you may consider useless or best removed.

Consider this: you may think as C++ as an OOP (Object Oriented Programming) language and *only* that and want to improve it for that. But that's short-sighted. If you do that, you *might* damage it for the people who think of C++ as a way for writing level-level kind of programs manipulating bits ... the C style. There has to be room in the world for all the choices. If we can't achieve that, it would be very unpleasant. C++ is a multi-faceted language and an engineering compromise. If people say "let's just remove feature so-and-so to get C++ better in area so-and-so" things might begin to fall apart. Chances are that this would create another tugging match between people each trying to "improve" the language to suit their own biases ... and only those biases. So, please think about what you're doing, what you're here for, and what C++ is about.

Another tension I've seen also is by people who want to maintain compatibility *within* their product line versus people who want to keep their programs compatible *across* machines. For instance, I like to run to run on a DEC, or a HP, or an IBM, or an Apple, or whatever ... preferable the same unmodified program. So, my bias is for compatibility across people's product lines.

But consider if you're a manager, managing all the languages of Sun or Apple or whoever. You want things to work nicely the other way: you want consistency in the way people work in different languages. That is a real and building tension. We have to look out for it and we have to avoid going into either ditch: that would cause a fight -- again -- between those two notions of compatibility.

What I'm trying to do here is to talk about some things I've noticed in working with C++ over the years. There will *always* be someone with a genuine need for the opposite bias. So, consider the C++ world en mass, not just the - relatively small - world you live in.

5. C++ Code Maintenance & Compatibility

[4] Tensions:
People with C++ code to maintain people
vs without C++ code to maintain
vs people with C code to maintain:
- C++ compatibility vs abstract elegance
- C compatibility vs C++ elegance
- existing implementations
  - cfront
  - Zortech
  - G++
  - Oregon
  - Greenhills
  - ...
  - (promise to break ALL compilers)

There will also be a tug-of-war between people that maintain C++ code and people who have C to maintain, as well as people who have neither. That is, in the later case, if you don't have any C++ code, you'll be quite willing to make a lot of incompatible changes to make the language better. On the other hand, if you have a couple of million lines of C++ code lying around, compatibility becomes

much more important. Similarly, people with a lot of C code and no C++ code yet will be willing to sacrifice current C++ features for C compatibility and incompatible improvements to the non-C features; there will be a tug-of-war here... and a lot of what we are saying and the work for the next years will be to balance these concerns.

So, I emphasize balance.

There's no simple right answer.

We have a lot of existing implementations. One of the things that is good about getting standardization started this early is that that implementations haven't departed too from the current manuals.

However, the language has not been sufficiently precisely defined for the various implementations to avoid minor incompatibilities. Let us not take one implementation, whichever it may be, and push that as defining the language. Let us not get into the trenches where *my* implementation is right and *your* implementation is wrong. But, instead, let's work together and try and write a manual which defines a standard that goes all the way.

My conjecture is that we should break all existing implementations. That's the only way of ensuring that we don't get into a commercial fight. I think all the compiler writers know that this is not an unreasonable thing to consider. Now don't misread me: I'm not saying let us go and change the syntax and semantics. Far from it. I'm saying let's remember that none of our implementations are perfect. Therefore, let's not get into a slinging match over which one is exactly right. None are.

6. Disasters <gulp!>

[5] Disasters:

One thing (just one!) YOU feel passionately about.
One thing to bring home to show that YOU caused
to be added.
    Pascal committee: Unanimity, say 90%.
If you'll vote for my feature I'll vote for yours.
    Just say no
Excitement! (remember 'noalias')
    Cooling off period - sub-committee for
    the evaluation of major features.

What I'm about to say now is one of the key things we need to talk about. At various times, I thought that standardization could be a great opportunity. It might even relieve me of a lot of work! But consider ...

Which are the ways we can go into the ditch?
Which are the disasters that we must start against?
How can we really spoil the language and past and future work?

The first disaster is really simple. Each one of you here has something that you feel passionate about ... one thing that's really important to you ... one thing that will may you feel really lousy if you have to go away from this standardization effort without getting it done. But there's 50 people or so in this room. Do you want 50 significant new features in C++?

You can kill a language that way

Please remember: if your really insist that you have to bring one thing home... just the one thing that's important... you really can kill a language. You can double the size of the language easily. This could have numerous, and irreparable side-effects. This could happen to any language. C++ is no exception.

Always keep in mind ANSI C. The committee is generally acknowledged as having done a superb job of avoiding creeping featurism in language design. But it's is also said that the average C compiler doubled in size

If *you* do that to C++ you get a **large** compiler.

If you do that, **and** throw in some good features and great ideas, you get something that will be a superset of ADA and PL/1.

So, please watch out, and please remember: it is a very dangerous attitude to come in feeling passionately about "just one thing" ... feeling that you MUST bring something home to the people that trust you, and to the people that respect you for being on the committee... and to people who pay your ticket.

BEWARE!

What we want is a coherent standard... something that is well defined... something that is well blended.... something you can live with. Any one features is less important than that. And especially: any 50 features are

Now remember, over the years a lot of people will get involved. There's 50 people here now. The total number of people that's going to be going through this committee is going to be 250? 300? 400? We must not fall in that trap. The total number of features that these 400 people feel passionately about could be VERY high: 50? 100? 400?

I was wondering how can we avoid that? One suggestion I had which I'd like you to think about is that addition and change ideally should be done unanimously. Voting rules can be adjusted to provide a reasonable and practical definition of "unanimous."

So, perhaps, it should be made reasonable hard to make a change. Because the main problem, the main danger to C++, is that it is a relatively young language and people have bright ideas about it. People don't agree exactly what C++ is and certainly not on what it really ought to be. We can be buried under an avalanche of improvements, so watch out.

Now the thing that I fear under all circumstances is an attitude like "I have to go home to prove to my manager and prove that I did something good here". I do. So do you. So, if you vote for my feature, I'll vote for yours! I don't know how to avoid this phenomenon. Please try and minimize such horse trading. All I can do is plea and warn. A good language isn't just a shopping list of nice features - but a shopping list of features is what you will get from such feature trading.

The thing that's even more dangerous than that, though, is excitement. Think of "noalias." This brings us back to ANSI C again. "Noalias" got introduced by the C committee. The C committee had a problem about what to do with the lack of protection against aliasing in C. This is a serious problem - especially to the writers and users of highly optimizing C compilers - and a good solution would be beneficial to everybody.

Now consider that the ANSI C committee was very well established and a very experienced committee. They understood the language and its underlying model. However, they got themselves excited about the 'noalias' solution. They wrote into the standard something that nobody was able to understand and nobody could agree on what the draft standards document really said. By doing this, the committee created a problem that undermined its authority and made C the laughing stock of quite a few communities.

They survived it; C can take a lot of abuse. It HAS in it's lifetime. So can C++, but we are not as well established a committee as the C committee was, C++ is not as established a language as C is, and there is much more for people to get excited over with C++. This really must not happen because we can make ourselves irrelevant. There has been enough standards made that are irrelevant because nobody took them serious. If we get excited and throw in features that can't be understood, are useless, or delays the standard for years, we are in deep trouble. Remember: *We need a standard. Soon.*

I was thinking, how can one avoid such problems? Perhaps we should institutionalize a kind of cooling off period. It is very hard to be excited for more than a month or so. People have different cooling down periods, but I suggest to seriously think about a procedural way of guaranteeing time for cooling down. Remember: haste makes waste.

The thing I thought about is that usually proposals come to the committee, they're circulated for a week or two, and people then go in and they argue about it for 2 minutes or 2 days. And then they vote.

Now, nobody wants to get excited in a way that gets them into trouble. But, if the ANSI C committee could get swept into mistakes by excitement then so can we. And easier.

So I suggest that we think about institutionalizing a cooling down period in of the form: if somebody declares a change major, it should be put to somebody to evaluate any impact on the rest of the language. That person will then give a report about the feature and its impact at the next meeting. Only then the committee can vote.

[6] Agreement:

Agreement is more important than perfection
But
don't agree to disagree
don't agree to vagueness
don't agree to "plain wrong"

As I mentioned earlier, in general, agreement is more important than perfection. I mean, consider: C++ is an engineering compromise. So are all languages. So is the Golden Gate Bridge. Being an engineering compromise is NOT contradictory to beauty. What we have to remember is not to go for a wild chase for perfection. Understand this.

On the other hand, we cannot just simply agree because we can. Agree on something sensible, *don't agree to disagree*. That is to say, well, I want feature X to work that way and you want feature X to work this way, so we'll specify 2 acceptable ways of doing X into the standard or leave it undefined. That doesn't makes sense, so, don't agree to disagree. That gets us nowhere except backwards ... which leads into the next point.

Don't agree on vagueness. The devious way of getting agreement by agreeing to disagree, is to ignore the fact that you say something that has 2 possible incompatible implementations.

At don't agree to something that's clearly plain wrong. I've seen all of these things in various language standards despite common sense, which is why I mention them. It HAS to be avoided.

7. Handling Language Extensions

[7] Language extensions
more than we can imagine
more than we can handle
=> PT
=> EH
GC
Meta classes
dynamic linking
concurrency

Another things I fear about standardization (and this has at least been my experience) is demand for new features. We haven't had a formal C++ users' group so I have acted as a clearing house for ideas over the past years. People with bright ideas, people who wanted to know what the features of the language were/are and people who couldn't find the answer in the manual, etc. have talked to me.

I *know* what is coming. I have lists of ideas that has come in over the years. We're going to get a fairly huge list of bright ideas. My attitude over the years has been to respond "No. What's the question?". Then I carefully store away the suggestion with the other information. I'm not being nasty, I'm just trying to arbitrate reasonably and not get rushed. There are more things that people want to do with C++ than we can imagine. There will be many more suggested improvements than we can handle, reasonably, and also integrate properly into the language. And we have to remember that these ideas I'm talking about are only the reasonable ones. So, the language can be **doubled or tripled** in size by only the reasonable ideas. Reasonable ideas are defined as something that will make somebody's life a whole lot better. If we took all of them we're finished.

If I have to priorities what people are talking about, the top of my list would be parameterized types (PT), exception handling (EH), garbage collection (GC), meta classes, dynamic linking, and concurrency. (Note: PT was presented in my PT paper published in various places. EH will be presented in Andy Koenig's and my EH paper at the USENIX conference this April).

Meta classes is word which covers a lot of things. My problem with metaclasses is, not that I dislike them, but, I've never met two people who agree what they really are or should be in the context of C++.

Dynamic linking of classes will be another major issue. Some people argue that special language features are needed to support dynamic linking. I don't, I think dynamic linking will be an example of a major "feature" that can be provided through libraries and without change to the language definition.

The question though is: Can you afford a feature?
Does it damage other features in the language?

In the case of PT and EH my feeling is that there is consensus from the community that they are needed. There may be a semi-consensus about how to do PT. And I hope we can reach some consensus on how to do EH. But it's not easy. Both of those features have the potential of breaking your programming language for lower-level programming and breaking your programming environment unless you buy the biggest box there is. So let's be careful.

[8] "Spirit of C++"

C++ is an engineering compromise
exists to solve problems
be wary of "perfection"
driven by constraints more than "principles"
orthogonality is a second order principle

In the C community, there is a lot of talk about the "spirit of C". I never quite understood it, since Dennis [M. Ritchie] was alive and well :-). It left me wondering what the spirit of C actually was. I realized that you *have* to define it because not everybody can just go and ask Dennis. Similarly, we will be faced with the problem of defining what the spirit of C++ is. That's especially important since we need some direction about which 'improvements' are within the scope of the language we're working on. So, I have a couple of comments trying to help those wondering what the spirit of C++ might be.

The first thing to remember is that C++ is an engineering compromise. There are people who think that 'engineering' and 'compromise' are dirty words and claim that they can do pure science. I don't think so. All languages are engineering compromises. And we should acknowledge that languages exist to solve problems, not to give good Phd talks... not to be able to write the most beautiful solution on a paper.

There are real problems out there and you must look at what the real problems are and try to solve them. In particular, you also have to limit it to the class of problems C++ was meant to solve.

For instance, a bridge may be a good bridge, but there is not one bridge that is suitable for all bodies of water. There is also a question of cost; you may be able to build the perfect bridge, but after a point, improvement for the sake of perfection becomes plain ridiculous and will in fact prevent people from crossing the water by delay or cost.

So, let's not try to solve *all* the problems. Let's not try to come up with the perfect programming language. Even if we started from scratch we would fail. So, once again, be wary of perfection. Perfection takes *a lot* of time and you may come up with something that in reality is unusable and is not that good.

C++, as designed, has been driven more by constraints than principles. What I mean is it has to solve problem X on machine Y under criteria Z of time and space. That's very important. Orthogonality is important, however it is not necessarily the primary principle in many cases. This is a conclusion I've had from years of looking at design problems. Be wary of the principle orthogonality; it can cause rampant growth in the size and complexity of compilers... You shouldn't accept a feature because "things fit better in the manual that way." I hope you *do* agree that compiler writers have a major role in languages - as well manual writers.

[9] "Spirit of C++"

what you don't use you don't pay for
don't leave room for lower level language
aim at radical portability
    of language
        implementations
        code
    of libraries
        interfaces
        maybe code

Another thing that I go back and look at when I look at a potential feature is "what you don't use, you don't pay for". Lots of languages give you lots and lots of goodies. However, what you as a programmer wanted was some very, tiny, little corner. You certainly don't want to pay for all of it, including the goodies you don't use. By and large, there is very few features in C++ that will do this to you. That means that you can do something like bit twiddling with maximum efficiency. The other features are totally independent. There are very few features in C++ that blow your objects up if you don't want them blown up to a large size by adding "house keeping" information. Similarly, nothing will eat up your CPU cycles "to help you" unless you somehow asked for it.

So, when you consider adding features, try not to raise the minimum cost of using existing features because of the impact of the new features.

Also, don't leave room for a lower level language underneath. What I mean is that you know what happens to higher level languages that can't do the whole job. People will start writing this little corner of the program in some lower level language, say C...., and next year the whole program is a C program.

I aim at radical portability of language implementations - at least of SOME language implementations. In particular, considering just code (and not libraries) for a moment, C++ has nice the property in common with C, that it needs only a minimal amount of run-time support to run. There is nothing in the semantics of C or C++ that require an elaborate 'abstract machine' or 'run-time environment' to run a minimal program.

Don't damage portability.

Libraries are very important, both in their existence and in their potential for portability. I see the language as a way of writing good libraries. It is the mechanism for writing good libraries. We need the libraries and the language's features can be used as a catalyst for getting such libraries created. Library interfaces should be very portable, implementations need not be although that would be nice. A lot of good libraries cannot be implemented portably, though. One of the advantages of C++ is that you *can* distinguish from the interface, which may be standard and portable, from the implementation, that cannot always be.

[10] "Spirit of C++"

Serve users first and compiler writers only second
    rely on compiler technology
    minimize run-time support
    distrust elaborate mechanisms

Now you must permit me to to speak as a compiler writer. The following principle is very dangerous in the hands of non-compiler writers, so please listen carefully: Basically the language is there to serve its users and *not* to serve its compiler writers. However, let's not do anything that makes the compiler

writers task impossible or really unnecessarily hard. We'll have to make that judgement.

But don't get me wrong. Of course, if there is consideration with merit and a choice between inconveniencing a dozen or two dozen compiler writers and inconveniencing a few thousand, perhaps 100 thousand users, well maybe it's time for the compiler writers to be inconvenienced. There will be compiler writers on this committee and they will probably scream when this principle is taken too far. I know that I will. But again, we need to standardize, and moreso, we need implementations that can be used and can be available.

C++ differs from a lot of 'modern' languages by relying heavily on compiler technology. C++ is designed on the assumption that a compiler is an important part of the programmers toolkit. Much work that is done by a C++ compiler is done at run time in other languages and systems. This is deliberate. The reason is partly to be able to provide strong guarantees about programs - early detection of errors - and partly to enable C++ programs to run effectively on conventional machines - even conventional machines with very few resources. There are lots of boxes out there. Requiring run-time support that is simple enough to work on just about any box is important.

Minimizing the runtime support is one of the principles of C++. I also have a general distrust of elaborate mechanisms. I'm not sure I understand them. I'm not sure I'm able to implement them well. I'm not sure I can work with them once I implement them. And furthermore, I'm not sure two compiler writers can implement them the same way.

[11] "Spirit of C++"

Don't break C or C++ compatibility
    unless you REALLY Have to
If anything major hasn't been used
    it probably doesn't work
'neat' isn't sufficient reason
    to introduce a feature
'but I need it' isn't sufficient reason
    to introduce a feature

We have a high degree of compatibility with C and with older version of C++. Don't try to break compatibility unless you *really* have to. That is, you should try very, very hard to stay compatible.

I think I tried harder than anybody else in the business to stay compatible, but sometimes you *have* to make the break otherwise something important gets damaged. So, again, it's where a balanced judgement comes in. It's not a hard and fast principle saying "no compatibility, 100% compatibility." Instead go for 99.99%" and deal with compatibility in a measured and rational manner.

Moving along, it's worth noting when considering new features and compatibility: if anything major hasn't been used, it probably don't work. That makes some good sense. Over the years, people from some C++ shops have gotten reputations for being rather conservative in compiler design. That may surprise some people watching C++ from the outside. But basically many things haven't been used, or tried. And in particular of those that have, many haven't been used by anybody but the builder. That doesn't mean it is no good - but most often it means that more work and thought is needed before a generally useful feature that fits correctly with the rest of the language is found. Take it easy when these situations come up. Similarly that something is "neat" is *not* a sufficient reason for introducing it into the language;

"I really need it" is not a sufficient reason either. Remember what I said earlier in this talk about this. There are so many people with so many genuine needs that is you try and serve them all, the language dies. In that case, nobody wins.

9. Standardization Worries

[12] Worries:

C++ young for a standard?
C++ users too inexperienced?
    C++ is not just C
    C++ is not Smalltalk
    C++ is not Ada
    C++ is not Lisp
    ...
    C++ is not silly putty
Are we too late?

I have some worries about the standardization of C++. For instance, is C++ too young for a standard? Is there sufficient agreement about what C++ is, and what it ought to be? Enough that we can actually get a standard in a reasonable time? Are the C++ users too inexperienced? **Including us** - the people standardizing it.

And then there's all the people who think C++ either is 'just C' or should be 'just C.' Also, C++ is not Smalltalk. There are fundamental differences between C and C++, as well as between C++ and Smalltalk, etc. And these fundamental differences *should* and had better be acknowledged. Realize that there are people who would like to use the name C++ - after completely changing the language. They may have good reasons. Such people need not be devious. They may simply feel that the RIGHT way of doing things is like THAT and not the way it is currently done in C++. We have to be careful here: **C++ is NOT silly putty.**

Another worry is that in some sense we can ask: Are we too late? Have too many different ideas about what C++ is or what it should be or what is should not be gone about so that is it now too late for us to reconcile them? Are there already too many dialects with 'interesting' features floating around on people's boxes? Are there already strong vested interests in this feature or that feature that we must extend the language in several mutually incompatible ways to please such vested interests?

10. History

[13] History:

use/experiment = >
    1.0/book
use + 1st external ref man review = >
    2.0/ATT reference manual draft ()
use + 2nd external ref man review = >
    My latest (manual with commentary)

The situation is that we've come to standardize C++. So, what do we have to build from in the sense of previous information? Well, there's a lot of user experience since the AT&T 1st release, cfront 1.0, the commercial release. There's also been a lot of experience since my book. So we grew for about four years and I realized that we'd hit standardization eventually and that it was time to start preparing. What we needed then was a good manual for review. There's a lot of people in this room who gave good feedback, input, and corrections to the current reference manual and I think the current reference manual - and the fact that it represent the work of MANY people - puts us in a reasonably good state.

The reference manual that AT&T sent out with release 2.0 was extraordinary in the sense that it had the word "draft" on it. The reason it had that was to acknowledge that the second external manual review was in progress and people had valid concerns and that they also have valid input through which a lot of valuable stuff has come into the language.

If you wonder what an external review looks like, this is the second one:

< < waving about 10 inches of paper in the air > >

it's about as thick as the first one. As the standards committee you're going to see stuff *much* bigger than this. But this load is what I've been under. The difference between the 2.0 reference manual and the new manual I'll be sending you is a about a half of year's work. This

< < waving a book manuscript in the air > >

is the Annotated Reference Manual with most of the comments there from previous input and I hope you will put it to good use.

BTW, one concern about standards is about copyright and such. Of course, AT&T will release the copyright of the C++ manual for ANSI to use for standardization soon so that there will be no legal problems.

11. Pleas

[14] Plea:

Standardize C++
Don't try to design D

And now the final plea, completely summarizing what I've said:

We need a standard.
We need it as soon as possible.
As such, if you start designing C++'99 or D or whatever you want to call it, we have failed.

So, Standardize C++ and don't try to design D.

Acknowledgements

Comeau Computing would like to thank Bjarne for his proper enlightenment of the committee members and for allowing us to distribute this document to enlighten future committee members and all other interested parties. In addition, Comeau Computing would not only like to thank William Mike Miller for allowing us to borrow his tape of the talk so that it can publically available, but also for his understanding when it was realized that the tape had been lost in transit to us (it did finally arrive, 28 days after postmark).

/* the end */