ISO Doc No: WG21/N0826
ANSI Doc No: X3J16/96-0008
Date: January 23, 1996
Reply To: Erwin Unruh
erwin.unruh@mch.sni.de

Recognizing non-standard C++

Erwin Unruh

Siemens Nixdorf Informatiossysteme AG
Department of Software Development Systems
C/C++ Front-End Laboratory LoB BS2000 SD 21
Otto-Hahn-Ring 6
D–81739 München
Germany

1 Introduction

When trying to write the header files for our compilers the question came up:

> *How do I tell whether the compiler runs in Cfront mode or standard mode?*

There was no direct answer since the WP does not support this question.

The macro `__cplusplus` does not fit this question. It is mainly used to distinguish C mode from C++ mode. It's value is unspecified in the WP so you cannot rely on it.

Other solutions introduced by some compiler may help on a specific platform, but they do not generalize. Some compilers define a macro for the compiler version, or a macro indicating the strictness option. Those macros are specific to a compiler and may be different when using another compiler. They may even differ whith a new version of the same compiler.

What I want to do is write a header file like

```
 /* header file for C and C++, standard and old version */

#if defined (__cplusplus)
  /* C++ mode */
  #if /* what do I write here ? */
    extern "C" int f(int) throw();
  #else
    extern "C" int f(int);
  #endif
#else
  /* C mode */
  #if defined(__STDC__) && __STDC__ +0 == 1
    int f(int);
  #else
    int f();
  #endif
#endif
```

2 Problem solved by ISO-C

ISO-C did solve this problem with the macro `__STDC__`, which was specified to have the value 1. There is the problem that non-conforming implementations had two possibilities, namely not defining the macro or defining it with the value 0. The rationale reserved the greater numbers for future use.

This was justified since the macro was invented by the committee and had no previous semantics. Market force prevented any abuse of the macro (like defining it without claiming conformance).

3 Possible solutions

We have several possibilities to tackle this problem. I see the following:

3.1 Ignore

Strictly speaking the behaviour of any non-conforming implementation is not part of the standard. So we can just say we don't need to do anything.

But ISO-C had tackled this problem. It may be outside the scope of the standard, but is a topic in real life C++ programming.

I do not think this is a viable option.

3.2 Specifiy `__cplusplus`

To solve the problem we can try to re-use the macro `__cplusplus`. We could specify a distinct value to indicate standard conformance.

The value of 1 is a bad solution since it is used in present compilers. Any other value may do, but it should have a meaning.

The best meaningfull value I have seen is the date of the standard. For details see section 3.3.2.

3.3 Specify an additional macro

The cleanest solution would be to specify a new macro which distinguishes standard C++ from pre-standard C++. The names used here are my suggestion, they can be changed to more speaking (or more or less obscure) names.

We have two main semantics of such a macro:

3.3.1 `__STDCplusplus__`

We could just duplicate `__STDC__` by defining an additional macro with a name similar to `__STDCplusplus__` which must have the value 1. We would than face the same problems ISO-C had, but we would know about the problems.

To avoid (part of) the problem, we could add a note or footnote that a non-conforming implementation should not define this macro. Then the line `#ifdef __STDCplusplus__` would recognize standard C++.

3.3.2 `__cplusplus_version`

We could look at the ISO-C amendment. There a macro `__STDC_VERSION__` which is defined to be the date of the amandment. So we would have a macro like `__cplusplus_version` with the value being the official date of the standard.

This would have the benefit that a future version of the C++ standard (or an amendment) would not have to introduce a new version macro. It could instead rely on this one.

We could also indicate that non-conforming implementations should not define this macro or that they should avoid certain values. We could say that the values from 1 to 99999L may be used by non-conforming implementations.

Conforming implementations would set this value to the year and month of the final adoption of this standard, which is scheduled for July 1997. So the value would be 199707L.

If the schedule changes, this value should be updated.

4 WP changes

I give WP text for the last three options. The first option does not need WP text.

4.1 for version 3.2

Change in 16.8 [cpp.predefined] paragraph 1 the entry for __cplusplus to:

> `__cplusplus` The name `__cplusplus` is defined to the value 199707L when compiling a C++ translation unit.
>
> Footnote: It is intended that future standards will replace the value of this macro by a greater value. Non-conforming compilers should use a value with at most five decimal digits.
>
> Box: The value is intended to be the date of final approval of this standard. If the schedule slips, the value of this macro should be adapted.

4.2 for version 3.3.1

Add in 16.8 [cpp.predefined] paragraph 1 the following entry:

> `__STDCplusplus__` The name `__STDCplusplus__` is defined to the value 1 when compiling a C++ translation unit.
>
> Footnote: Non-conforming compilers should not define `__STDCplusplus__`.

4.3 for version 3.3.2

Add in 16.8 [cpp.predefined] paragraph 1 the following entry:

> `__cplusplus_version` The name `__cplusplus_version` is defined to the value 199707L when compiling a C++ translation unit.
>
> Footnote: It is intended that future standards will replace the value of this macro by a greater value. Non-conforming compilers should use a value with at most five decimal digits.
>
> Box: The value is intended to be the date of final approval of this standard. If the schedule slips, the value of this macro should be adapted.