Doc No: **X3J16/95-0095**
WG21/N0695
Date: Thu, May 18, 1995
Project: Programming Language C++
Reply to: John Micco
(johnm@bedford.symantec.com)

Proposed Changes to C++ <stdarg>

Introduction

This document describes some small changes to the `va_start()` macro from `<cstdarg>` that are required in order to make the C language standard argument support work properly for C++. The current definition of `va_start()` has some small wording problems when used in the context of C++.

ANSI C standard va_start

In Austin, the core working group considered whether to allow references as the last parameter prior to the ellipsis in a variable argument list function. In doing so we discussed what limitations C had placed on variable argument list functions in order to be somewhat consistent. The C standard does not restrict (in the language) the use of any argument type prior to the ellipsis of a variable argument list function. For example:

```
void f(char c, ...);     // Legal in ANSI C
void f(int i,...);       // Legal in ANSI C
```

At issue is the handling of the `va_start()` macro in many implementations of ANSI C. The `va_start()` macro is typically implemented as:

```
#define va_start(p, arg)            p = (va_list) (&arg+1)
```

The argument `arg` to the `va_start()` macro is the argument prior to the ellipsis in the function parameter list. Many implementations of ANSI C promote this argument using the default integral promotions. This led to the ANSI restriction that the second argument to `va_start()` be a type that would result after applying the default integral promotions.

The exact wording of the restriction from the ANSI C standard (X3.159-1989) section 4.8.1.1 p. 123 is:

"The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the ...). If the parameter *parmN* is declared with the register storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined."

The rationale for ANSI C does not explain the prohibitions on this parameter, but I believe these restrictions are made in support of the common implementation of the va_start macro given above. The restrictions are:

- The parameter cannot be declared with the `register` storage class. In ANSI C, the address of variables declared with the register storage class cannot be taken, hence this restriction.

- The size of the parameter must match the size of its declared type. This leads to the prohibition on arrays, functions, and non-promoted arguments. For example consider:

```
void f(char x, ... )
{
      va_list ap;
      va_start(ap,x);    // Expands to:
                         //    ap = (va_list) (&x+1)
}
```

  Given this definition of the `va_start()` macro, implementations which promoted `x` before calling the function `f()` would not calculate the address of the first variable argument properly[1].

C++ problems with va_start

The ANSI C definition for va_start() presented above is clearly not sufficient for C++. There are several problems with it:

- The address of register variables can be taken in C++. This means that the prohibition on the

[1] Keep in mind that this is not merely a concession to existing implementations of K & R style C, many hardware implementations (including the MC68000) have data alignment requirements that require the stack be aligned and would not be able to pass a char type parameter in a single byte.

register storage class for C++ is superfluous.

- C++ does not define the "default argument promotions", and therefore the concept must be extrapolated from the ANSI C definition.
- Reference types in C++ violate the unstated rule of the ANSI C standard that the size of the parameter must match its declared type. For example:

```
void f(double &dr, ...)
{
        va_list ap;
        va_start(ap, dr);        // Expands to:
                                 // ap = (va_list) (&dr+1)
}
```

This would fail to yield the address of the variable arguments on most existing implementations of C++.

Proposed wording for va_start in C++

In order to resolve these small problems and enable the existing implementation of `va_start()` to continue to work for C++, I would like to propose the following addition to [lib.support.runtime] section 18.7 in the current draft:

> The restrictions placed upon the second parameter to the `va_start()` macro in `<cstdarg>` should be replaced by:
>
> The parameter parmN is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the ...). If the parameter parmN is declared with a function, array or reference type, or with a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is undefined.
>
> See Also: [expr.call], ISO C subclause xxx.

One problem that I noticed with the existing standard is that the concept of passing an argument for which there is no parameter (an ellipsis argument) is not given a name in the C++ standard. I believe this process (called default argument promotion in ANSI C) is the correct restriction for the va_start() marco in C++, and the editor may wish to provide a name for this concept so that the wording in clause 18.7 can be more concise.

This wording allows existing implementations of va_start to continue to work properly without adding any severe additional restrictions.

Summary

The ANSI C restrictions on the second argument to va_start() were not quite appropriate for C++, the problem areas identified were:

- Use of “default argument promotions” from the ANSI C standard that does not exist as a term in C++.
- Superfluous prohibition on the register storage class.
- Reference types cause problems for most implementations of variable arguments and do not agree with the underlying intent of the ANSI C standard

These problems were addressed by rewording the restriction on this argument and adding a new prohibition on reference types.