Accredited Standards Committee*
X3, INFORMATION PROCESSING SYSTEMS

Doc No: X3J16/91-0060
Date: July 8,1991
Project: Programming Language C++
Ref Doc: X3J16/91-0059
Reply to: Jonathan E. Shopiro
shopiro@research.att.com

Report on Changes in the June '91 Working Paper

Introduction

Changes between X3J16/91-0059, the June 1991 Working Paper, and the previous version, X3J16/91-0009, are shown by change bars in the draft. The slides from my talk on the new draft, given in Lund, Sweden, are attached to this report.

The most important changes in this revision of the draft are the sections on the C++ memory model and the definitions of terms in Chapter 1. The C++ memory model is essentially the same as that of C, but there is no single clear definition of the C memory model in the C Standard. Instead, it is defined primarily by implication.

Memory Model and Type

There are two notions of type in both C and C++. The first is the type of an object, which depends on the language construct (declaration, new-expression, etc.) that caused the object to be created. The second is the type of an expression used to access an object or a location in memory. Of course the type of the expression used to access an object is usually the same as the type of the object and in that case there are no semantic problems. However many questions about the freedom of implementors can be asked as questions about what must or may happen if an object of type *A* is accessed by an expression of type *B*. Thus a clear memory model helps the user know what he can rely on and lets the implementor know what he must do.

A perhaps questionable issue is whether the fundamental unit of storage in the model is the byte or character. The C Standard 3.1.2.5, Types, discusses types without mentioning bytes. However, C2.2.1.2, Multibyte characters, makes it clear that characters are stored in bytes.

A related issue is addresses and pointers. There is no direct representation for the address of a byte in C or C++, but C3.5.2.1, Structure and Union Specifiers, discusses the addresses of structure members as byte addresses. C1.6 says that an object occupies a contiguous sequence of bytes. The phrase "contiguous sequence of bytes" is not defined but the simplest way to explain it is that it is a set of bytes, each of which has an address and the addresses of the bytes in the sequence are linearly ordered.

A pointer is not the same as a byte address because it has a type. However, a pointer to void is essentially a byte address because a pointer of any type can be converted to pointer to void and back without change. Also the definition of alignment (C1.6) indicates that a byte address is essentially an integer.

* *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 311 First St. NW, Suite 500, Washington, DC 20001

I think that what all this implies is that the byte address of a C object is the address of the first byte of the object. The Standard would be a lot simpler if it came out and said so. For C++ the same thing could be true for complete objects, that is, objects which are not sub-objects of some other object.

Definitions

The definition of *undefined* is intended to show that the Standard defines the C++ language and specifies what conforming C++ processors must do. Certain operations in the C++ language are errors and for some of those, the Standard does not specify what the processor must do. Those cases are called "undefined" and the processor may do anything, as far as the Standard is concerned.

Similarly, the definitions of *unspecified*, *implementation-defined*, and *locale-specific* are intended to constrain the behavior of the processor a little more tightly than in the C Standard.