N641/97-004
03-Jan-97

Implementation-defined Integral Types

Randy Meyers
Digital Equipment Corporation

1. Introduction

The good news is that this paper is not a proposal to add any new features to Standard C. This paper

- does not add any new ways to declare types
- does not add any new types to Standard C
- does not add specification based integer types
- does not add range based types.

The bad news is that this paper does argue that the committee should reverse a three year old answer to a defect report.

Defect Report 067 (reproduced below in Section 5) asked if an implementation can add additional types (e.g., **`__int64`**) to the type categories (such as integral type) in the Standard, and if so, can such implementation extension types be used to define typedefs in the Standard library. The committee made three points in its answer:

1. An implementation may not add new (implementation extension) types to the Standard type categories.
2. Implementation extension types may not be used to declare any typedef in the Standard headers.
3. No strictly conforming program can use an implementation extension type for any purpose.

This paper proposes the following alternative interpretation of the C89 Standard (which should carry over into C9x):

1. A conforming implementation may add implementation extension types to the Standard type categories provided the extension types have all of the properties of the existing types in that category. For example, an implementation defined integral type must use a binary number representation and must be a valid operand of any operator that operates on integral types.
2. An implementation defined integral type may be used to declare any typedef in a Standard header whose type is specified as being an integral type.
3. No strictly conforming program can directly use the name of an implementation defined type. However, a strictly conforming program can use an implementation defined type via a Standard typedef from a Standard header.

This paper will show that the previous interpretation has bad consequences, that it is unreasonable, and that the revised interpretation is a reasonable alternative.

2. Bad consequences of previous interpretation

2.1 Effects on existing implementations

The previous interpretation probably renders non-conforming any implementation with an extension type. All of the operators in the language have constraints that limit their operands to type categories in the standard. Is it really useful to make shifting a **`long long`** or **`__int16`** value be a constraint violation because it does not have integral type? Off hand, I don't know of any implementation that takes the position that every operation on an extension type is a constraint violation. I do not believe that the programmer is helped by making every operation on an extension type a constraint violation.

2.2 Portability suffers

The previous interpretation seems to prohibit **`inttypes.h`** from defining any typedefs using extension types, which limits the header's usefulness in many situations that it could promote portability. For example, some vendors of 64 bit machines have proposed making **`int`** and **`long`** be 64 bits, and **`short`** be 32 bits. A 16 bit integer type would not be available using a Standard C keyword, but could be made available using typedefs in inttypes.h.

Another example: A manufacturer of a 36 bit machine might add **`__int8`**, **`__int16`**, and **`__int32`** types to ease porting programs tightly tied to 32 bit hardware. The typedefs in **`inttypes.h`** are a natural way to make such types available even though **`char`**, **`short`**, and **`int`** might have sizes in bits of 9, 18, and 36.

Programs that really need a particular sized integer (or integer with particular properties) are better served by allowing **`inttypes.h`** to provide the type even if the type is not mapped to a Standard C keyword.

2.3 Discriminates against economically important environments

The previous interpretation prohibits for the C89 Standard an economically important growth path for 32 bit operating systems running on 64 bit hardware. Thirty-two bit operating systems wish to maintain the "traditional" mapping of Standard C data types on 32 bit machines (16 bit **`short`**s; 32 bit **`int`**s and **`long`**s) for compatibility, but wish to take advantage of the large address space of 64 bit machines. The natural way to accomplish this is to allow 64 bit pointers, and allow **`size_t`** and **`ptrdiff_t`** to be a 64 bit integers provided via an extension like **`long long`** or **`__int64`**. (Since C9x has added **`long long`**, this problem will not surface for the new standard until the address space grows beyond 64 bits.)

3. Is the previous interpretation reasonable?

"If it looks like a duck, quacks like a duck, and swims like a duck, maybe its a duck"

The previous interpretation is certainly difficult to explain to programmers since it draws a fairly artificial distinction between the Standard types and extension types. The effect is not unlike a Zen koan:

- This implementation extension type that holds integers is not an integral type
- This implementation extension type that is just like an integral type except it is different, but not in any practical way
- It is "wrong" (a constraint violation) to shift this implementation extension integer type, or add it, or use it as an array index, but that is how you are expected to use it

The simplest way to explain implementation extension types, and the simplest way to program using them, is to call them integral types (or floating types, or pointer types, as appropriate).

4. The new interpretation is reasonable

4.1 Does merely adding a extension type alter the behavior of a strictly conforming program?

An extension can be added to a conforming implementation provided it does not alter the behavior of any strictly conforming program. An implementation can add new types and meet that requirement.

First, assume that the implementation gets the basics right. The implementation does not add any new keywords whose spelling is in the user namespace, or if such keywords are added, they are protected by pragmas or the magic "include a non-Standard header" technique. The implementation does not change any of the promotion or type balancing rules for expressions not involving the new type. The implementation does not interpret constants with defined values for the Standard types any differently.

If an implementation gets the above basics right, programs written only using the Standard types are interpreted the same with or without the extension. Thus, as long as the implementation does not introduce a use of an extension type into a program that is strictly conforming, the extension is suitable for a conforming implementation. (The implementation might introduce the extension into a strictly conforming program by using the extension for a Standard typedef, such as **`size_t`**. That case is discussed in Section 4.2.) Since the extension does not alter the behavior of a strictly conforming program, there is no harm in letting the extension type also be a member of a Standard type category, such as integral type or arithmetic type.

4.2 Does using an extension type for a standard typedef alter the behavior of a strictly conforming program?

There are a number of typedefs in the Standard whose types are specified as being an implementation defined type from a particular type category, for example, size_t is an unsigned integral type. For such types, a strictly conforming program is limited in what it can assume about such times.

For example, a strictly conforming program can assume that it can add two values of type **`size_t`** because the addition operator is defined for integral types. It can also assume that it can use a value of type **`size_t`** as an array index because the **`[]`** operator is defined for integral types. However, it can not assume that **`size_t`** is the same type as **`unsigned char`**, or **`unsigned short`**, or **`unsigned int`**, or **`unsigned long`**, since the type is free to be any of those types.

In terms of the above assumptions, an extension unsigned integral type could be used for **`size_t`** since the extension type could be used anywhere a Standard integral type could be used (when you don't know which Standard integral type you are dealing with): the same operators that you can use on a Standard integral type work with an extension integral type.

However, some people have argued that it is important to only permit Standard types to be used to defined Standard typedefs in order to allow programmers to assume that **`unsigned long`** is guaranteed to be large enough to hold any value of type **`size_t`**. This property is exploited when careful programmers are using **`scanf`** and **`printf`** to read and write **`size_t`**. The idiom is:

```
printf("Sizeof array = %lu\n", (unsigned long) sizeof array);
```

There are a few problems with this:
1. The Standard never explicitly says that **`long`** and **`unsigned long`** are the largest integral types and that no larger types are permitted. Given that **`long long`** predates the Standard by about five years, inventing the principle in an interpretation seems unreasonable.
2. Adding **`long long`** to C9x means that **`unsigned long`** is not the longest type anymore.
3. Although the above idiom has many benefits, it is rare. Most programs fail to put in the cast and just use the %d or %ld format.
4. Allowing larger types doesn't strictly break the above idiom, it at most limits its usefulness to objects whose size is smaller than 2^{32} bytes in size.
5. The typedefs and format macros in inttypes.h for the largest unsigned integer provide a much cleaner and upward compatible way of writing the idiom.

4.3 Can a strictly conforming program use an extension?

Normally, no. The definition of strictly conforming is in fact designed to prohibit the use of any extensions. However, there is a long standing principle in the Standard that strictly conforming programs can use extensions indirectly through Standard defined interfaces.

Consider **`offsetof`**. The offsetof macro expands to an integral constant expression yielding the offset of a member within a struct type. The committee has long acknowledged that constant expressions must be extended by an implementation in order to support offsetof. Typical extensions include adding a built-in function that returns the offset or allowing expressions like **`(int) &((struct T *)0)->member`**. Such extensions, if they directly appeared in a user program would prevent the program from being portable (no two implementations need implement the same extension to support offsetof). However, if the extension is used indirectly through the Standard-defined interface **`offsetof`**, the program can be strictly conforming (all implementations have to implement **`offsetof`** in some fashion).

Another example is **`strcpy()`**. The Standard allows implementations to provide a macro for **`strcpy()`** in **`string.h`** that expands into inline code for a machine-specific string instruction. If a user program directly used such an extension, the program would not be strictly conforming. However, using the extension indirectly though the Standard defined interface **`strcpy()`** allows the program to be strictly conforming.

In a similar fashion, a program that uses a implementation extension integral type through a Standard typedef for an integral type may be strictly conforming.

5. Defect Report 067

Defect Report #067
Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question

The terms "signed integer type," "unsigned integer type," and "integral type" are defined in subclause 6.1.2.5. The C Standard also uses the terms "integer type," "signed integral type," and "unsigned integral type" without defining them. Integer-valued bitfields are also introduced in subclause 6.5.2.

a) For each of the following types, which if any of the six categories above do they belong to?

```
char
signed char
unsigned char
signed short
unsigned short
signed int
unsigned int
signed long
unsigned long
int : N             /* i.e. bitfield of size N */
signed int : N
unsigned int : N
```
enumerated type

b) For each of these categories, do the **`const`** and/or **`volatile`** qualified versions of the types belonging to the category also belong to the category?
c) Can an implementation extension add other types defined by the C Standard to any of these six categories?
d) Can an implementation define other types (e.g. **`__very long`**) which belong to any of these six categories?

e) If the answer to (c) or (d), or both, is yes, can **`size_t`** and **`ptrdiff_t`** be one of these other types, or must it be a type in the above list?

Response

a) “Signed integer type”, “unsigned integer type”, and plain “integer type” are used interchangeably with “signed integral type”, “unsigned integral type”, and “integral type” in the C Standard. This observation makes it easy to categorize the types in your list.
b) Yes, see subclause 6.1.2.5.
c) No, the list in the C Standard is meant to be exhaustive. For example, **`float`** cannot be defined as an integer type.
d) No strictly conforming program could contain an instance of such a type. The treatment of such types is beyond the scope of the C Standard.
e) No, it must be a type in the list. For example, **`size_t`** cannot be defined as unsigned **`__int24`**.

6. Defect Report 067 (Early Response)

The response in Section 5 is the official response published in the Record of Response. Jutta Degener found an earlier version of the response in her notes that shows that originally the committee was somewhat more flexible on this topic: Implementations could have additional integral types, but could not use such types for standard typedefs (like **`size_t`**).

(The discussions of DR 067 seem to have occurred at the Kona and San Jose meetings in 1994.)

Suggested Response

a) signed integer type, unsigned integer type, and plain integer type are used interchangeably with signed integral type, unsigned integral type, and integral type in the C Standard. This observation makes it easy to categorize the types in your list.
b) Yes, qualification produces a version of a type, not a separate type.
c) and d) No, the list in the C Standard is meant to be exhaustive. For example, "four" signed types cannot be read as "four or more". Conforming implementations may add other distinct types (such as **`__int24`**) to these categories, but must not use such types where a standard-specified type is required. For example, **`size_t`** cannot be defined as **`unsigned __int24`**.