The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

Copyright ©2002-2008 Derek M. Jones. All rights reserved.

6.7.2 Type specifiers

type specifier syntax

1378

```
type-specifier:
        void
        char
        short
        int
        long
        float
        double
        signed
        unsigned
        _Bool
        _Complex
        _Imaginary
        struct-or-union-specifier
        enum-specifier
        typedef-name
```

Commentary

declaration syntax

The syntax of *type-specifier* does not enumerate all possible combinations of keywords that may be used to specify an arithmetic type. The syntax for *declaration-specifiers* supports the occurrences of more than one *type-specifier* in a single declaration.

The wording was changed by the response to DR #207.

C90

Support for the *type-specifiers* **_Bool**, **_Complex**, and **_Imaginary** is new in C99.

C++

The nonterminal for these terminals is called *simple-type-specifier* in C++ (7.1.5.2p1). The C++ Standard does contain a nonterminal called *type-specifier*. It is used in a higher-level production (7.1.5p1) that includes *cv-qualifier*.

The C++ Standard includes **wchar_t** and **bool** (the identifier bool is defined as a macro in the header **stdbool.h** in C) as *type-specifiers* (they are keywords in C++). The C++ Standard does not include **_Bool**, **_Complex** and **_Imaginary**, either as keywords or type specifiers.

Other Languages

Equivalent *type-specifiers* seen in other languages include: **integer**, **short real**, **real**, **double precision**, **boolean**, **character**, **ptr**, and **complex**.

Common Implementations

Implementations add additional type specifiers, as extensions, either to support some special purpose hardware functionality (e.g., the Intel 8051 processor[2] has an area of storage containing what are known as *special function registers*). Some vendors (e.g., Keil,[5] Tasking[1]) have added type specifiers (e.g., **sfr** and **sbit**) to provide a mechanism for declaring objects denoting locations in this storage area, or to give special semantics to objects that are application specific.

Motorola added some vector functionality (what it calls AltiVec Technology Resources[6]) to its implementation (the MPC7400) of the IBM PowerPC instruction set. This enabled 128 bits of storage to be treated as either: sixteen 8-bit objects, eight 16 bit objects, or four 32-bit objects. Their translator for this processor supports the keyword **__vector** as an extension. This keyword acts as a type specifier, indicating that for instance an object is composed of 16 unsigned chars. Operations on and between objects declared with this type specifier operate on all of the subobjects at the same time. For instance, an add operation will add

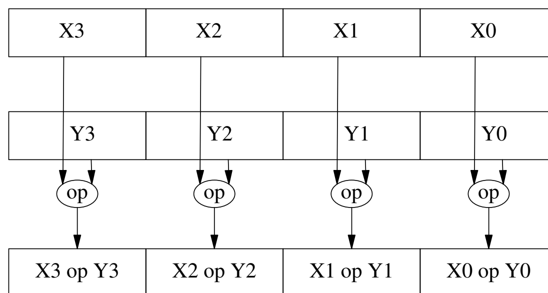

Figure 1378.1: Behavior of packed single-precision floating-point operations supported by the Intel Pentium processor.[3]

corresponding unsigned chars from each of the two operands. The streaming SIMD extensions (SSE)[3] to the Intel x86 instruction set support a similar set of operations (see Figure 1378.1) on blocks of 128 bits and a few translators[4] make use of them.

Coding Guidelines

A guideline recommendation dealing with the use of a subset of the available type specifiers is discussed elsewhere.

?? object int type only

Table 1378.1: Common token pairs involving a *type-specifier*. Based on the visible form of the .c files. The type specifiers **_Bool**, **_Complex**, and **_Imaginary** did not appear in the visible form of the .c files.

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
unsigned long	38.7	72.2	**; long**	0.1	6.2
unsigned short	5.8	63.8	**, void**	0.3	5.8
char *p	74.5	63.3	**static unsigned**	3.8	5.5
(signed	0.0	60.5	**extern double**	1.3	5.5
; enum	0.1	45.5	**} int**	2.2	5.3
(struct	2.9	41.8	**{ signed**	0.0	5.2
; float	0.1	40.0	**static char**	4.1	5.1
; union	0.0	33.7	header-name **double**	0.2	5.1
static void	33.7	32.7	**static struct**	6.4	4.8
(float	0.0	32.0	**register enum**	1.6	4.6
(unsigned	1.0	29.0	**long *p**	7.1	2.8
(void	1.4	26.6	**int** identifier	87.6	2.3
; unsigned	1.0	26.4	**extern void**	21.5	2.1
; int	2.5	24.8	**struct** identifier	99.0	1.9
(char	1.0	23.9	**extern int**	32.1	1.7
{ union	0.0	23.4	**short *p**	21.8	1.4
(double	0.0	22.9	**register struct**	19.1	1.4
; double	0.0	19.8	**const unsigned**	6.2	1.4
void *p	17.5	19.0	**const struct**	11.1	1.3
, unsigned	0.6	18.9	**typedef struct**	62.4	1.2
} void	4.1	18.0	**register int**	23.0	1.2
unsigned char	21.2	18.0	**register char**	10.2	1.2
; struct	1.3	17.6	**volatile unsigned**	25.6	1.1
; char	0.8	17.5	**void** identifier	61.7	0.9
, int	1.4	15.9	**void)**	17.5	0.9
static int	28.2	15.1	**register unsigned**	6.1	0.9
; signed	0.0	14.7	**extern char**	7.4	0.9
{ struct	4.3	14.5	**const void**	5.3	0.8
identifier **double**	0.0	13.1	**signed short**	11.3	0.7
{ unsigned	1.9	12.5	**int)**	6.6	0.6
, struct	0.8	12.2	**extern struct**	6.9	0.5
{ int	4.8	11.5	**volatile struct**	15.5	0.4
{ enum	0.1	11.1	**long** identifier	68.3	0.4
typedef union	3.2	11.0	**long)**	21.7	0.4
(short	0.0	11.0	**float *p**	9.2	0.3
; short	0.0	10.6	**char** identifier	22.6	0.3
(int	1.0	10.6	**typedef unsigned**	6.2	0.2
, float	0.0	10.6	**signed long**	20.8	0.2
const char	54.1	10.4	**double *p**	7.9	0.2
{ float	0.1	10.2	**volatile int**	7.4	0.1
(union	0.0	9.9	**unsigned** identifier	7.0	0.1
, char	0.4	9.9	**union {**	34.5	0.1
(long	0.2	9.2	**signed char**	22.6	0.1
, enum	0.0	9.2	**short** identifier	60.9	0.1
unsigned int	24.6	9.1	**enum {**	13.4	0.1
{ double	0.0	8.6	**union** identifier	65.5	0.0
typedef enum	10.8	8.2	**signed int**	7.5	0.0
, double	0.0	8.2	**signed)**	37.9	0.0
int *p	4.1	8.1	**short)**	14.0	0.0
, union	0.0	8.0	**float** identifier	64.3	0.0
, signed	0.0	7.9	**float)**	26.1	0.0
) enum	0.0	7.1	**enum** identifier	86.6	0.0
{ char	1.3	7.1	**double** identifier	70.7	0.0
static signed	0.0	6.5	**double)**	19.1	0.0
; void	0.3	6.3			

Constraints

1379 At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name.

declaration at least one type specifier

Commentary

C no longer supports any form of implicit declaration.

A *specifier-qualifier-list* is the production used in the syntax for structure and union specifiers. It denotes an optional list of *type-specifier* and *type-qualifier*.

struct/union syntax

C90

This requirement is new in C99.

In C90 an omitted *type-specifier* implied the type specifier **int**. Translating a source file that contains such a declaration will cause a diagnostic to be issued and are no longer considered conforming programs.

C++

> *At least one `type-specifier` that is not a `cv-qualifier` is required in a declaration unless it declares a constructor, destructor or conversion function.*[80)]

7.1.5p2

Although the terms used have different definitions in C/C++, the result is the same.

Other Languages

Most languages require that declarations contain some form of type specification (although some of them will create an implicit declaration if an identifier is referenced without first being declared). Some languages have no means of explicitly declaring the type of an object. For instance, ML implementations need to deduce the type of an object from the context in which it is used. Fortran uses the first character of an identifier to implicitly give it a type, if it is not given one explicitly through a declaration. Identifiers starting with any of the letters `I` through `N`, inclusive, having integer type and all other identifiers having real type. There is even an **`IMPLICIT`** statement that allows the developer to control the types implicitly chosen for identifiers starting with different alphabetic letters.

Common Implementations

Most C90 implementations do not issue a diagnostic for the violation of this C99 constraint. It is expected that most C99 translators will continue to treat such declaration as implying the type **int**.

Coding Guidelines

Many developers will continue to use C90 translators for years to come and these will not check for a violation of this constraint. Some static analysis tools do not check source for violations of constraints on the basis that this check is performed by translators (thus reducing the amount of work that implementors of such tools need to do). The existing C90 behavior is well defined, experienced developers will be familiar with it, and it is simple for developers new to C to learn. Occurrences of an omitted type specifier in existing code is rare. Given this rarity and the small cost paid by readers of the source (often a slight confusion causing a task switch followed by the realization that the **int** *type-specifier* has been omitted).

cognitive switch

1380 Each list of type specifiers shall be one of the following sets (delimited by commas, when there is more than one set on a line);

type specifiers sets of

Commentary

A set is an unordered collection of items (tokens in this case). The syntax permits an arbitrary long sequence of different type specifiers. This constraint delimits the set of possible type specifier combinations that can appear in a conforming program.

Other Languages

Few languages support the large number of different combinations of type specifiers available in C. Algol 68 was unusual in that its use of a two level grammar permitted families of an infinite number of types, e.g., **int**, **short int**, **short short int**, **short short short int**, . . . , and **int**, **long int**, **long long int**, **long long long int**, and so on (implementations were allowed to specify a *shortest* and *longest* type, after which additional **short** and **long** had no affect on representation).

Coding Guidelines

If developers followed the principle of least effort, when typing source, we would expect the type specifiers most often used to be the first one listed on each line and the ones least often used to be the last on the line. With one exception existing code appears to overwhelmingly use the shortest form (see Table 1378.1). The exception is the type **unsigned int**. Developers appear to use the **unsigned** type specifier for all unsigned types. While developers will be most practiced using the shorter forms, the additional cost of using any of the longer forms is likely to be small. For this reason a guideline recommendation would not appear to have a worthwhile cost/benefit.

---

type specifiers ordering

the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers. 1381

Commentary

Future language directions

There is no future language direction specifying any change in this behavior.

Other Languages

Most languages require a fixed order for type specifiers. Even though they may contain of more than one token, they are usually treated, by the syntax, as if they were a single token. Very few languages allow other declaration specifiers to be intermixed with type specifiers.

Coding Guidelines

Existing code rarely contains a list of type specifiers having an order that is not one of those specified in the C Standard (see Table 1378.1). Based on this common practice in existing code the possibility that readers will only read what they consider to be the minimum number of tokens needed to deduce an objects integer type has to be considered. For instance, readers may believe that the declaration `long unsigned id_name` declares an object to have type **long**.

Cg 1381.1

If more than one type specifier occurs in a declaration the order used shall be one of those listed in the Standard.

---

type specifiers possible sets of

1382

```
—  void
—  char
—  signed char
—  unsigned char
—  short, signed short, short int, or signed short int
—  unsigned short, or unsigned short int
—  int, signed, or signed int
—  unsigned, or unsigned int
—  long, signed long, long int, or signed long int
—  unsigned long, or unsigned long int
—  long long, signed long long, long long int, or signed long long int
—  unsigned long long, or unsigned long long int
—  float
—  double
```

—~ **long double**
—~ **_Bool**

—~ **float _Complex**
—~ **double _Complex**
—~ **long double _Complex**
~~—~**float _Imaginary**~~
~~—~**double _Imaginary**~~
~~—~**long double _Imaginary**~~
—~ struct or union specifier
—~ enum specifier
—~ typedef name

Commentary

The keyword **signed** can only make a difference to the type in two cases (depending on implementation-defined behavior it may not make any difference), (1) when it appears with **char**, and (2) when it appears in the declaration of a bit-field. However, support for its use creates a symmetry with the keyword **unsigned**.

char range, representation and behavior
1387 bit-field int

Rationale

Some pre-C89 implementations allowed type specifiers to be added to a type defined using **typedef**. Thus

```
typedef short int small;
unsigned small x;
```

would give x the type **unsigned short int**. The C89 Committee decided that since this interpretation may be difficult to provide in many implementations, and since it defeats much of the utility of **typedef** as a data abstraction mechanism, such type modifications are invalid.

The wording was changed by the response to DR #207.

C90

Support for the following is new in C99:

— **long long**, **signed long long**, **long long int**, or **signed long long int**
  — **unsigned long long**, or **unsigned long long int**
  — **_Bool**
  — **float _Complex**
  — **double _Complex**
  — **long double _Complex**

Support for the *no type specifiers* set, in the **int**, **signed**, **signed int** list has been removed in C99.

```
1  extern x;   /* strictly conforming C90  */
2              /* constraint violation C99 */
3  const  y;   /* strictly conforming C90  */
4              /* constraint violation C99 */
5         z;   /* strictly conforming C90  */
6              /* constraint violation C99 */
7         f(); /* strictly conforming C90  */
8              /* constraint violation C99 */
```

C++

The list of combinations, given above as being new in C99 are not supported by C++.

Like C99, the C++ Standard does not require a translator to provide an implicit function declaration returning **int** (footnote 80) being supplied for a missing type specifier.

Other Languages

Most languages use only one sequence of tokens to represent a given type and have a relatively small number of different integer types (often only one). Cobol allows the number of digits used in the representation of decimal types to be specified and many Fortran implementations support more than one integer type. Languages that support integer subranges allow millions of integer types to be defined, although implementations invariably map these to the same underlying representations that are used by C implementations. PL/1 uses a precision specifier rather than keywords to indicate the number of bits (or digits) in various types (e.g., `BINARY(16)` specifies a binary (integer) type represented in nine bits).

Common Implementations

Support for **long long** existed in some vendors implementations for a number of years before C99 was ratified. The set **short double** was supported as a synonym for **float** in a few prestandard implementations. Some implementations continue to support this set for backwards compatibility.

Table 1382.1: Occurrence of *type-specifier* sequences (as a percentage of all type specifier sequences; cut-off below 0.1%). Based on the visible form of the `.c` files.

Type Specifier Sequence	%	Type Specifier Sequence	%
int	39.9	**long**	2.2
void	24.3	**unsigned**	1.6
char	15.6	**unsigned short**	0.9
unsigned long	6.2	**float**	0.6
unsigned int	4.0	**short**	0.5
unsigned char	3.4	**double**	0.5

The type specifiers ~~_Complex and _Imaginary~~ shall not be used if the implementation does not provide ~~those~~ complex types.[102] 1383

Commentary

This is a constraint that depends on a feature being available in the implementation being used.

The wording was changed by the response to DR #207.

C90

Support for this type specifier is new in C99.

C++

Support for these type specifiers is new in C99 and are not specified as such in the C++ Standard. The header **<complex>** defines template classes and associated operations whose behavior provides the same functionality as that provided, in C, for objects declared to have type **_Complex**. There are no equivalent definitions for **_Imaginary**.

Other Languages

implementation two forms complex types

Cobol and Fortran (77) support optional type specifiers. Only a few languages support complex types.

Coding Guidelines

Source code that uses the specifier **_Complex** or **_Imaginary** has a dependence on implementations providing the necessary support. Translating source code containing one of these type specifiers, using an implementation that does not support them, will result in a diagnostic being issued. The behavior is either defined, or a diagnostic is issued. There is no benefit from having a guideline recommendation.

Semantics

1384 Specifiers for structures, unions, and enumerations are discussed in 6.7.2.1 through 6.7.2.3.

Commentary

See subclauses 6.7.2.1 through 6.7.2.3 for the discussion.

struct/union syntax
enumeration specifier syntax
type contents defined once

1385 Declarations of typedef names are discussed in 6.7.7.

Commentary

See subclause 6.7.7 for the discussion

typedef name syntax

1386 The characteristics of the other types are discussed in 6.2.5.

Commentary

See subclause 6.2.5 for the discussion

types

1387 Each of the comma-separated sets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **int** designates the same type as **signed int** or the same type as **unsigned int**.

bit-field int

Commentary

That is, the sets appearing on the same line designate the same type.

The treatment of bit-field types is not quiet the same as the treatment of character types, although it has been influenced by similar historical factors (variations between early implementations). While an **int** bit-field is the same type as either the signed or unsigned forms, the type **char** is a different type from the signed and unsigned character types (although it is capable of representing the same range of values as one of them).

character types
char range, representation and behavior

C90

> *Each of the above comma-separated sets designates the same type, except that for bit-fields, the type **signed int** (or **signed**) may differ from **int** (or no type specifiers).*

C++

Rather than giving a set of possibilities, the C++ Standard lists each combination of specifiers and its associated type (Table 7).

Other Languages

Languages that offer some mechanism for controlling the number of storage bits allocated to an object, do not usually allow that mechanism to change the underlying type of the object.

Common Implementations

Many translators support an option that enables the developer to control how this choice is made.

Coding Guidelines

Most operands having a bit-field type promote to the type **int**. The only bit-field type that does not is one declared with the type specifier **unsigned int**, using a width that is the same as that of the type **unsigned int** (which can occur in source designed to be ported to a variety of architectures through the use of macros to defined the field width). While occurrences of operands having a bit-field type may promote to **int**, the underlying value representation supports a limited range of values. It is possible that translating the same source using different implementations will result in a dramatic change in the range of representable values (in the case of signed to unsigned, negative values being unrepresentable).

Cg 1387.1

The declaration of a bit-field type shall always include one of the type specifiers **`signed`** or **`unsigned`**.

footnote 102

102) ~~101)Implementations are not required to provide imaginary types.~~ Freestanding implementations are not required to provide complex types. 1388

Commentary

conforming freestanding implementation

Complex types are one of many features a freestanding implementation does not have to provide. Not being able to accept a program that uses such types does not affect such an implementation's conformance status.

The wording was changed by the response to DR #207.

C90

Support for complex types is new in C99.

C++

There is no specification for imaginary types (in the **`<complex>`** header or otherwise) in the C++ Standard.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.7). 1389

References

1. Altrium BV. *C166/ST10 v8.0 C Cross-Compiler User's Guide*. Altrium BV, 2003.
2. Intel. *MCS 51 Microcontroller Family User's Manual*. Intel, Inc, 272383-002 edition, Feb. 1994.
3. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.
4. Intel. *Intel C++ Compiler User's Guide*. Intel, Inc, 2002.
5. Keil. *C Compiler manual*. Keil Software, Inc, ??? edition, May 2005.
6. Motorola, Inc. *AltiVec Technology Programming Interface Manual*. Motorola, Inc, 1999.