The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.2.5 Types

types

The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. 472

Commentary

Without a type, the object representation is simply a pattern of bits. A type creates a value representation from an object representation.

Other Languages

There is a standard for data types: *ISO/IEC 11404:1996 Information Technology— Programming languages, their environments and system software interfaces – Language-independent datatypes*. Quoting from the scope of this standard "This International Standard specifies the nomenclature and shared semantics for a collection of datatypes commonly occurring in programming languages and software interfaces, . . . ".

Some languages, for instance Visual Basic, tag objects with information about their type. When accessed, the implementation uses the tag associated with an object to determine its type, removing the need for the developer to explicitly specify a type for an object before program execution. Some languages (e.g., Algol 68 and CHILL) use the term *mode* rather than *type*.

Coding Guidelines

representation information using effective type ??

Accessing the same object using more than one type is making use of representation information and is discussed elsewhere.

(An identifier declared to be an object is the simplest such expression; 473

Commentary

primary-expression syntax integer constant type first in list

Such an expression is a *primary-expression*. Both the form and value of an integer constant determines its type, so an integer constant is not simpler than an identifier.

---

the type is specified in the declaration of the identifier.) 474

Commentary

declaration at least one type specifier

C99 does not support the implicit declaration of any identifier. Labels may be used before they are defined, but they must still be defined.

Common Implementations

The behavior of some implementations on encountering an identifier that has not been declared is to provide a default declaration (as well as issuing the required diagnostic). Declaring the identifier to be an object of type **int** often prevents cascading diagnostics from being generated. A more sophisticated error-recovery strategy is to examine the token immediately to the left of identifier (adding the identifier as a member of the structure or union type if it is a selection operator).

---

types partitioned object types incomplete types

Types are partitioned into *object types* (types that fully describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes). 475

Commentary

This defines the terms *object types*, *function types*, and *incomplete types*. Function types can never be transformed into another type, although a pointer-to function type is an object type. When an incomplete type is completed, it becomes an object type. In the case of array types it is possible for its complete/incomplete status to alternate. For instance:

```
1  extern int ar[];   /* ar has an incomplete type here.  */
2
```

```
3  void f_1(void)
4  {
5  extern int ar[10]; /* ar has a complete type here.    */
6  }
7                     /* ar has an incomplete type here. */
```

Structure and union types can only be completed in the same scope as the original, incomplete declaration.

C++

Incompletely-defined object types and the void types are incomplete types (3.9.1). 3.9p6

So in C++ the set of incomplete types and the set of object types overlap each other.

An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not a ***void*** *type.* 3.9p9

A sentence in the C++ Standard may be word-for-word identical to one appearing in the C Standard and yet its meaning could be different if the term *object type* is used. The aim of these C++ subclauses is to point out such sentences where they occur.

Other Languages

Many languages only have object types. Some languages support pointers to function types, but don't necessarily refer to a function definition as a function type. Incomplete types, in C, were originally needed to overcome the problems associated with defining mutually recursive structure types. Different languages handle this issue in different ways. The Java class method can be viewed as a kind of function type.

---

476 An object declared as type **_Bool** is large enough to store the values 0 and 1.

_Bool large enough to store 0 and 1

Commentary

Many existing programs contain a type defined, by developers, with these properties. The C committee did not want to break existing code by introducing a new keyword for an identifier that was likely to be in widespread use. **_Bool** was chosen. The header **<stdbool.h>** was also created and defined to contain more memorable identifiers. Like other scalar types, **_Bool** is specified in terms to the values it can hold, not the number of bits in its representation.

While an object of type **_Bool** can only hold the values 0 and 1, its behavior is not the same as that of an **unsigned int** bit-field of width one. In the case of **_Bool** the value being assigned to an object having this type is first compared against zero. In the case of a bit-field the value is cast to an unsigned type of the appropriate width. An example of the difference in behavior is that even, positive values will always cause 1 to be assigned to an object of type **_Bool**, while the same values will cause a 0 to be assigned to the object having the bit-field type. The standard does not prohibit an object of type **_Bool** from being larger than necessary to store the values 0 and 1. The only way of storing a value other than 0 or 1 into such a *larger object* depends on undefined behavior (i.e., two members of a union type, or casting pointer types). Whether, once stored, such a value may be read from the object using its **_Bool** type will also depend on the implementation (which may simply load a byte from storage, or may extract the value of a single bit from storage).

The term *boolean* is named after George Boole whose book, "An investigation of the laws of thought",[4] introduced the concept of Boolean algebra (or *logic* as it is commonly known, a term that is something of an overgeneralization).

C90

Support for the type **_Bool** is new in C99.

C++

3.9.1p6 *Values of type* **bool** *are either* **true** *or* **false**.

4.5p4 *An rvalue of type* **bool** *can be converted to an rvalue of type* **int**, *with* **false** *becoming zero and* **true** *becoming one.*

9.6p3 *A* **bool** *value can successfully be stored in a bit-field of any nonzero size.*

Other Languages

Many languages, including Java, support a boolean type; however, they don't usually get involved in specifying representation details. Fortran calls its equivalent type **LOGICAL**.

Common Implementations

Only a few processors contain instructions for loading and storing individual bits— from/to storage. Most implementations have to generate multiple instructions to achieve the required effect. Representing the type **_Bool** using a byte of storage simplifies (shorter, faster code) the loading and storing of its values. Many implementations make this choice.

The Intel 8051[15] has a 16-byte internal data-storage area and supports bit-level accesses to it. Several implementations (e.g., Keil,[19] Tasking[1]) include support for the type specifier **bit**, which enables objects to be declared that denote specific bits in this data area.

Coding Guidelines

George Boole intended his work to " . . . investigate the fundamental laws of those operations of the mind by which reasoning is performed.". It was some time before people realized that the reasoning carried out by the human mind is based on principles other than mathematical logic.

operand
convert automatically

Experience with character types has shown that developers sometimes overlook the effects of promotion, of operands, to the type **int**. It remains to be seen whether objects of type **_Bool** will be used in contexts where such oversights, if made, will be significant.

Existing programs that define their own boolean type often use the type **unsigned char** as the underlying representation type. Given that implementations are likely to use this representation, internally, for the type **_Bool**, there would not appear to be any worthwhile benefits to changing the underlying type of the developer-defined boolean type. However, translators and static analysis tools are becoming more sophisticated and use of the type **_Bool** provides them with more tightly specified information on the range of possible intended values.

There are 10 kinds of developers, those that understand boolean roles and those that don't.

The type **_Bool** is based on concepts derived from boolean algebra and mathematical logic. Other than defining this type, the concept is not discussed again in the standard. Some languages (including C++) specify the result of some operators as having a boolean type. This is not the case in C99 (largely for backwards compatibility with C90).

logical negation
result type

While the C Standard does not use **_Bool** in the specification of any other constructs, the concept of a boolean role is invariably part of a developer's comprehension of source code. Other terms used by developers when discussing boolean roles include *flag*, *indicator*, *switch*, *toggle*, and *bit*. These terms are usually associated with situations having two different states, or values. True and false, or 0 and 1, are simply different ways of representing these states. Mathematically the representation values could just as well be 222 and 4,567, or 0 and any number except 0. However, in practice the representation does play a part and the minimalist approach is often used (the values 0 and 1 are also representable a bit).

bit

Should the concept of boolean be interpreted in its broadest sense, with any operation that can only ever deliver one of two results be considered as boolean? Or should the C definition of the type **_Bool** be used as the sole basis for how the concept of boolean is interpreted? The former interpretation probably corresponds more closely with how developers think about boolean concepts, while the latter avoids subtle problems with different representations of the two values used in the representation. However, there is no evidence to suggest that either interpretation is unconditionally better than the other.

At the source code level the difference in specification of behavior for boolean and the other arithmetic types can be a small one. In all but the first case C defines the result of the following operations have an integer type (which differs from C++ where some of them have type **bool**):

- Cast of an expression to type **_Bool**.
- The result of a relational, equality, or logical operator.
- The definition of an enumerated type containing two enumeration constants.
- The definition of a bit-field of width one.
- The result of the remainder operator when the denominator has a value of 2.

Is there a worthwhile benefit in a guideline recommendation that classifies certain kinds of operations as delivering a result that has a boolean role and places restrictions on the subsequent use of these results?

The type **_Bool** is new in C99, and there is little experience available on the kinds of developer comprehension costs associated with in its use. However, a boolean type has long been supported in other languages. Is there anything to be learned from this other language usage? The other languages containing a boolean type, known to your author, do not promote operands (having this type) to an integer type. Because the operands retain their boolean type, some of the operations (e.g., addition) available to the C developer are not permitted (without explicitly casting to an integer type). Creating a stricter type system for C which treated the type **_Bool** as being distinct from the integer types and specified a result type of **_Bool** for some operators (e.g., logical negation, relational, equality, logical-AND, and logical-OR), is likely to be a nontrivial exercise that will touch nearly all aspects of the language specification. The approach taken in these coding guidelines is to introduce the concept of roles (leaving the C type system alone) and provide guideline recommendations on the contexts in which objects having these roles may occur.

typedef
is synonym
logical
negation
result is
relational
operators
result type
equality
operators
result type
&&
result type
||
result type
object
role
boolean role
identifier
selecting spelling

An object having an integer type integer type has a boolean role if it is only expected to hold one of two possible values or it is assigned objects that have a boolean role.

The semantic associations implied by boolean usage could suggest spellings of identifiers denoting states that only ever take on one of two values. This issue is discussed elsewhere.

Example

```
#include <stdio.h>

extern _Bool E;

void f(void)
{
if (((E ? 1 : 0) != (!(!E))) ||
    ((E += 2) != 1)          ||
    ((--E, --E, E) != E))
   printf("This is not a conforming implementation\n");
}

void g(void)
{
_Bool Q;
struct {
```

```
17          unsigned int u_bit :1;
18        } u;
19  #define U (u.ubit)
20
21  Q = 0;     U = 0;   // sets both to 0
22  Q = 1;     U = 1;   // sets both to 1
23  Q = 4;     U = 4;   // sets Q to 1, U to 0
24  Q = 0.5;   U = 0.5; // sets Q to 1, U to 0
25  Q++;       U++;     // sets Q to 1; sets U to 1-U
26  Q--;       U--;     // sets Q to 1-Q; sets U to 1-U
27  }
```

---

char
hold any member of execution character set
byte
addressable unit
CHAR_MIN
CHAR_MAX

An object declared as type **char** is large enough to store any member of the basic execution character set. 477

Commentary

This requirement on the implementation is not the same as that for byte. One is based on storage and the other on type. The values of the CHAR_MIN and CHAR_MAX macros delimit the range of integer values that can be stored into an object of type **char**.

C++

3.9.1p1 *Objects declared as characters (**char**) shall be large enough to store any member of the implementation's basic character set.*

character
single-byte

The C++ Standard does not use the term character in the same way as C.

Other Languages

Many languages have some form of type **char**. Objects declared to have this type are capable of representing the value of a single character.

Coding Guidelines

character
single-byte

The terms *char* (a type) and *character* (a bit representation) are often interchanged by developers. While technically they have different meanings in C (but not C++), there does not appear to be anything to be gained by educating developers about the correct C usage.

---

basic character set
positive if stored in char object

If a member of the basic execution character set is stored in a **char** its value is guaranteed to be ~~positive~~non-negative. 478

Commentary

basic character set
may be negative
basic character set
fit in a byte

This guarantee only applies during program execution (i.e., it does not apply during preprocessing). The nonnegative value is also representable in a byte. No standardized character set specifies a negative value for its member representations. However, this is not a requirement on the character set an implementation may use. It is a requirement on an implementation that its representation of the type **char** be capable of supporting its chosen basic execution character set.

The wording was changed by the response to DR #216.

C++

The following is really a tautology since a single character literal is defined to have the type **char** in C++.

3.9.1p1 *If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character.*

The C++ does place a requirement on the basic execution character set used.

2.2p3

For each basic execution character set, the values of the members shall be non-negative and distinct from one another.

Other Languages

Few languages say anything about the values of character set members. But they invariably use the same character sets as C, so the above statement is also likely to be true for them.

Common Implementations

Although the basic execution character set has less than 127 members, it is possible for a character set to use values outside of the range 0 to 127 to represent members (and EBCDIC does).

Coding Guidelines

Dev ??

A program may rely on the value of a member of the basic execution character set stored in an object of type **char** being positive.

---

479 If any other character is stored in a **char** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.

Commentary

This is a requirement on the implementation. They cannot, for instance, specify the implementation-defined value of `'@'` as 999 when objects of type **char** can only represent values between -128 and 127 (but such an implementation-defined value would be possible if the type **char** supported a range of values that included 999).

An implementation supporting a character set containing a member whose value was greater than 127, assuming an 8-bit **char**, would have to chose a representation for the type **char** that had the same range as type **unsigned char**. Such character sets occur within mainstream European languages. The numeric values assigned, by ISO 10646, to characters in the ISO 8859–1 (Latin-1) standard fall in the range 32 to 255. ISO 8859

C90

*If other quantities are stored in a **char** object, the behavior is implementation-defined: the values are treated as either signed or nonnegative integers.*

The implementation-defined behavior referred to in C90 was whether the values are treated as signed or nonnegative, not the behavior of the store. This C90 wording was needed as part of the chain of deduction that the plain char type behaved like either the signed or unsigned character types. This requirement was made explicit in C99. In some cases the C90 behavior for storing *other characters* in a **char** object could have been undefined (implicitly). The effect of the change to the C99 behavior is at most to turn undefined behavior into implementation-defined behavior. As such, it does not affect conforming programs.

516 char range, representation and behavior

Issues relating to this sentence were addressed in the response to DR #040, question 7.

C++

The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific. 2.2p3

The only way of storing a particular character (using a glyph typed into the source code) into a char object is through a character constant, or a string literal. glyph

An ordinary character literal that contains a single `c-char` *has type* **`char`**, *with value equal to the numerical value of the encoding of the* `c-char` *in the execution character set.*

2.13.4p1 *An ordinary string literal has type "array of n* `const char`*" and static storage duration (3.7), where n is the size of the string as defined below, and is initialized with the given characters.*

3.9.1p1 *If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character.*

Taken together these requirements are equivalent to those given in the C Standard.

Other Languages

Few language specifications get involved in the representational details of character set members. In most strongly typed languages character literals have type **char**, so by definition they can be represented in a **char** object.

representation information using ??

Coding Guidelines

The guideline recommendation dealing with the use of representation information is applicable here. An example of a situation where implicit assumptions on representation may be made is array indexing. A positive value is always required, and the guarantee on positive values only applies to members of the basic execution character set.

Example

```
1  char ch_1 = '@';
```

Usage

In the visible form of the `.c` files 2.1% (`.h` 2.9%) of characters in character constants are not in the basic execution character set (assuming the Ascii character set representation is used for escape sequences).

---

standard signed integer types

There are five *standard signed integer types*, designated as **`signed char`**, **`short int`**, **`int`**, **`long int`**, and **`long long int`**. 480

Commentary

This defines the term *standard signed integer types*. Having five different types does not mean that there are five different representations (in terms of number of bits used). Multiple integer types, based on processor implementation characteristics, are part of the fabric of C. An implementation could choose to implement all types in 64 bits. They would still be different types, irrespective of the underlying representation. On processors that do not support 32- or 64-bit (needed for **`long`** and **`long long`** respectively) integer operations in hardware, there is a strong incentive not to use types requiring this number of representation bits.

floating types 497 three real

There are three real floating types.

C90

Support for the type **`long long int`** (and its unsigned partner) is new in C99.

C++

Support for the type **`long long int`** (and its unsigned partner) is new in C99 and is not available in C++. (It was discussed, but not adopted by the C++ Committee.) Many hosted implementations support these types.

Other Languages

Many languages only support a single integer type. Some implementations of these languages, for instance Fortran, include extensions that allow the size of the integer type to be specified. Fortran 90 contains the intrinsic function SELECT_INT_KIND which enables the developer to specify the range of powers of 10 that an integer should be able to represent. The intrinsic SELECT_REAL_KIND allows the decimal precision and exponent range to be specified.

The Ada Standard gives permission for an implementation to support the optional predefined types **SHORT_INTEGER** and **LONG_INTEGER**. Java contains the types **short**, **int**, and **long**. It also specifies that **long** is represented in 64 bits, so there is no need (at least yet) to specify a **long long** that may be larger than a **long**.

Common Implementations

There are commonly at least four bit widths used to represent these types— 8, 16, 32, and 64. On processors that do not support a 64-bit data type (needed for the type **long long**) an implementation will perform 64-bit operations via calls to internal library functions. Some processors do not even support 32-bit operations. Like the 64-bit case, the support is provided via calls to internal library functions.

Coding Guidelines

The fact that there are at least five (four in C90) integer types in C, and not one, is a cause of great misunderstanding, by developers and unintentional behavior, in programs. The existence of these different types causes implicit conversions to become an important issue.

operand convert automatically

There is a simple solution: Restrict programs to using a single integer type. The argument against this solution is often based on a perceived lack of resources (additional time to execute a program and greater storage requirements). The quest for efficiency is often uppermost in developers' minds when deciding on which integer type to use. For instance, an object taking on a small range of values, less than 128 say, is almost automatically given a character type. On RISC- and Intel Pentium-based processors the time taken to load or store an 8- or 32-bit quantity is usually the same. Once the value is loaded into a register, it is often operated on as a 32-bit quantity. There are 8-bit operations available on the Pentium, but they are not often generated by translators because of the extra complication needed for what can be small savings.

On the low-cost processors found in many freestanding environments, there are often resource limitations. Use of a narrower integer type can result in worthwhile performance improvements and storage savings. Implementations for such environments usually use the same representation for the types **int** and **short**. In a hosted environment the representation for the type **int** is usually the same as for the type **long**. The choice of representation for the type **int** is intended to be natural to the execution environment.

485 int natural size

Rev 480.1 The type specifiers **char**, **short**, and **long** shall not appear in the visible source, or be used via developer-defined macros or via developer-defined typedef names.

Dev 480.1 When resources are constrained and where a worthwhile benefit has been calculated a character type whose promoted type is **int** may be used.

Dev 480.1 If a program needs to represent an integer value that is outside the representable range of the type **int** the type **long** or **long long** type may be used.

Dev 480.1 If a program needs to represent an integer value that is outside the representable range of the type **int**, resources are constrained and where a worthwhile benefit has been calculated, the type **unsigned int** may be used.

Developers rarely have any control over the contents of system headers. These may contain typedefs, for instance size_t, that have an integer type other than **int**. While use of these system-defined typedefs has many advantages, they do provide an alternative route for operands having different integer types to appear within expressions.

wide character constant
type of
sizeof
result type
pointer
subtract
result type
integer types
sizes

It is possible for values of different types to occur in an expression, other than via accesses to declared objects. For instance, an operand might be a wide character constant, the result of the **sizeof** operator, or the result of subtracting two pointers.

The minimum range of values that can be represented in these integer types is specified by the standard. An algorithm may depend on some object being able to represent a particular range of values. Use of a typedef name, in declarations of objects, provides a single point of control for the underlying representation (the typedef name definition). The issue of encoding this range information in the typedef name is discussed elsewhere.

typedef
naming conventions

Some coding guideline documents recommend that these keywords not be used directly in declarations (e.g., MISRA rule 13). Instead, it is recommended that typedef names denoting integer types of specific widths be defined and used. Such typedefs can be applicable in some situations. However, the number of programs where it is necessary to be concerned about the widths of all integer object representations is small. A much more important association is implied by the semantics associated with typedef names, this is discussed elsewhere.

MISRA

typedef
assumption of no integer promotions
typedef name
syntax

Usage

It is possible to specify many of the integer types, in C, using more than one sequence of keywords. Usage information on integer types is given elsewhere (see Table **??**).

(These and other types may be designated in several additional ways, as described in 6.7.2.) 481

Commentary

type specifier
syntax

Useful for developers who are lazy typists, entrants to the Obfuscated C contest,[2] and writers of coding guideline recommendations who want to pad out their material.

Other Languages

Most languages define a single way of specifying every type.

---

extended signed integer types

There may also be implementation-defined *extended signed integer types.*[28)] 482

Commentary

The Committee recognized that new processors are constantly being developed to fill niche markets. These markets do not always require the characteristics normally expected of a general-purpose processor. Sometimes special-purpose uses result in unusual architectural designs. Rather than have vendors extending the language in different kinds of ways, the Committee has attempted to provide a standard framework for extended integer types. The **<stdint.h>** header is one place where these extended integer types may be defined.

Is using " . . . implementation-defined extended signed integer types." making use of an extension or is it implementation-defined behavior? It is listed as an implementation-defined behavior in annex J.3.5.

footnote 512
34

An implementation may also define new keywords to denote the existing types specified in the standard.

C90

The C90 Standard never explicitly stated this, but it did allow extensions. However, the response to DR #067 specified that the types of `size_t` and `ptrdiff_t` must be selected from the list of integer types specified in the standard. An extended type cannot be used.

C++

The C++ Standard does not explicitly specify an implementation's ability to add extended signed integer types, but it does explicitly allow extensions (1.4p8).

Other Languages

The Pascal/Ada language family allows developer-specified subranges (lower and upper bounds on the possible values) of integer types to be defined. These subranges provide information to the translator that enable it to select the appropriate underlying, processor, integer type to use.

Coding Guidelines

If the guideline recommendation dealing with using a single integer type is followed, any extended types will not occur in the source. 480.1 object int type only

Having occurrences of a nonstandard type, supported by a particular implementation, scattered throughout the visible source code creates portability problems. A method of minimizing the number of visible occurrences is needed. Use of a typedef name does not necessarily offer the best solution:

```
1  /*
2   * Assuming vendor_int is a keyword denoting an extended
3   * integer type supported by some implementation.
4   */
5  #if USING_VENDOR_X
6  typedef vendor_int INT_V;
7  #else
8  typedef int INT_V;
9  #endif
10
11 unsigned INT_V glob; /* Syntax violation. */
```

C syntax does not allow a declaration to include both a typedef name and another type specifier. Using a macro name to cover the implementation-defined keyword would avoid this problem. type specifier syntax

---

483 The standard and extended signed integer types are collectively called *signed integer types*.[29)] signed integer types

Commentary

This defines the term *signed integer types*. Unlike the integer types, other types do not have unsigned versions (floating-point types are always signed) and are not thought of in terms of being signed or unsigned. Hence the term commonly used is *signed types* rather than *signed integer types*.

C90

Explicitly including the extended signed integer types in this definition is new in C99.

---

484 An object declared as type **signed char** occupies the same amount of storage as a "plain" **char** object.

Commentary

The amount of storage occupied by the two types may be the same, but they are different types and may be capable of representing different ranges of values. The representation of the sign in the type **signed char** is part of the object representation and is not permitted to consume additional bits. 515 character types

---

485 A "plain" **int** object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range **INT_MIN** to **INT_MAX** as defined in the header **<limits.h>**). int natural size

Commentary

A "plain" **int** is the most commonly used type in source code. The choice of its representation has many important consequences, particularly in the quality of machine code produced by a translator. There may be several interpretations of the term *natural size* for some processor architectures. Efficiency is usually a big consideration. Execution-time performance efficiency is not always the same as storage efficiency for a given architecture. An implementors choice of "plain" **int** also needs to consider its effect on how integer types with lower rank will be promoted.

The choice of representation need not be decided purely on how the available processor instructions manipulate their operands. In function calls the most commonly passed argument type is "plain" **int**. The organization of the function call stack is an important design issue. In some cases it may have already been specified by a hardware vendor.[12, 28–30, 33] In this case the choice of "plain" **int** representation has already been dictated for all translators targeting that environment. ABI

Other Languages

Most languages say nothing about the representation of integer types; it is left to implementation vendors to decide how best to implement them for a given host. Java is designed to be portable across all environments. This is achieved by specifying the representation of all types in the language standard.

Common Implementations

Historically the "plain" **int** type was usually the same size as either the types **short** or **long**. It is rare to find an **int** type having a size that is between the sizes of these two types. This existing practice has affected existing code, which is often found to contain implicit assumptions about various integer types having the same widths.

width integer type

Coding Guidelines

Developers sometimes want the program they write, or at least large parts of them, to be independent of the host processor (the current market dominance of the Intel/Microsoft platform has caused many development groups to lose interest in portability to other platforms). Having a fundamental integer type whose representation depends on natural features of the host environment does not sound ideal. One solution is to hide implementation decisions behind a typedef name and recommend that developers use this rather than the **int** keyword (moving source code to a new implementation then only requires one change to the typedef definition).

It used to be the case that both 16- and 32-bit processors were commonly encountered in a hosted environment. The migration, for hosted environments, to 32-bit processors is now an established fact. The migration to processors where 64-bit integers are the natural architectural choice has only just started (in 2002). Whether it is yet worth investing effort on enforcing a guideline that recommends using a typedef name rather than the **int** keyword is uncertain (for programs aimed at the desktop environment).

Mobile phones and hand-held organizers are starting to support the execution of programs downloaded by their users. To conserve power and reduce costs, some of the processors used in such devices have a natural integer size of 16 bits. The extent to which a large number of programs will need to be ported from a 32-bit environment to these hosts is unknown.

The processors used in freestanding environments vary enormously in their support for different sized integer types. On some, support for a 16-bit **int** (the minimum standard requirement) does not come naturally. At the time of this writing a large amount of the software written for these processors does not get ported to other processors, it is device-specific (as much driven by the characteristics of the application as by the cost of changing processors). But developer organizations do want to reuse code if possible— it can reduce time to market (but not always total cost). In this development environment, where the size of the type **int** is likely to vary, there are obvious benefits in a guideline recommendation to use a typedef name.

object 480.1 int type only

The guideline recommendation dealing with using a single integer type specifies the use of the type **int**.

---

signed integer corresponding unsigned integer

For each of the signed integer types, there is a corresponding (but different) unsigned integer type (designated with the keyword **unsigned**) that uses the same amount of storage (including sign information) and has the same alignment requirements. 486

Commentary

This is another case of C providing a construct that mirrors a data type, and associated operations, commonly found in hardware processors. The unsigned integer types can generally represent positive values twice as great as their corresponding signed counterparts.

Other Languages

Few languages support unsigned integer types. Modula-2 uses the term *cardinal* to denote its unsigned integer type. Support for what were called *modular* types was added to Ada 95.

Coding Guidelines

The mixing of operands having unsigned and signed integer type in an expression is a common cause of unexpected program behavior. Developers overlook the effects of performing the integer promotions clauses

integer promotions

and usual arithmetic conversions, or are simply unaware that the operands have different signedness. This issue is also discussed elsewhere.

usual arithmetic conversions
operand convert automatically
480.1 object int type only

Following the guideline recommendation that only the type **int** be used implies that no objects having an unsigned type are declared. Does a deviation from this guideline recommendation have a worthwhile benefit? Measurements show (see Table 486.1) that unsigned types are much more common in embedded software than signed types. The fact that some languages do not have an unsigned type might be more of an indication of the applications written using them than that programs don't need to use such a type. The following are several ways of thinking about unsigned types in relation to signed types:

integer types sizes

integer promotions

- Signed types are the *natural type* to use and any usage of unsigned types needs to be justified. In many programs the range of values manipulated is well within the representable range of the integer types used. The ability of signed types to represent negative values frees developers of the need to think about the possibility of having to handle negative values created as the intermediate step during the evaluation of an expression. The type **int** is the type to which narrower types are converted by the integer promotions. The type **int** is equivalent to the type **signed int**. Signed types appear to be the type of least effort (for developers).
- Unsigned types are the *natural type* to use and any usage of signed types needs to be justified. Most quantities that are counted or measured have positive values (mathematicians refused to acknowledge the existence of negative quantities for many centuries[5]). In many cases applications do not deal with negative quantities. Unsigned types appear to share the same *positive quantity* semantics as applications.
- Neither type is considered more natural to use than the other. There does not appear to be any situation where this would be the default position to take.

Using the type **unsigned int** only prevents signed/unsigned conversion issues from arising if it is the only type used. Objects declared using a type of less rank will be promoted to the type **int**.

Alt 480.1 All objects declared in a program shall have an unsigned type.

Developers often only consider the use an unsigned type when the possible range of values being stored is not representable in the corresponding signed type. There may not be a signed type capable of representing the desired range of values (support for the type **long long** is new in C99), or use of this type may be considered to be *inefficient*. Whether signed/unsigned conversion issues are minimized by declaring all objects to have either a signed or unsigned type will have to be decided by the developer.

Usage

Usage information on integer type specifiers is given elsewhere (see Table **??**, which does not include uses of integer types specified via typedef names).

Table 486.1: Occurrence of objects having different width integer types (as a percentage of all integer types) for embedded source and the SPECint95 benchmark (separated by a forward slash, e.g.,embedded/SPECint95). Adapted from Engblom.[6]

	8 bits	16 bits	32 bits
unsigned	70.8/1.3	14.0/0.4	2.1/44.9
signed	2.7/0.0	9.4/0.3	1.0/53.1

A study by Engblom[6] compared the use of integer types in embedded source and SPECint95 (see Table 486.1). There are a number of possible reasons of Engblom's results. Most of the difference is due to the use of the type **unsigned char**. There can be significant performance advantages in using this type (because of processor characteristics <host processors, introduction>). There is also the observation that measurements of physical quantities are usually positive quantities. Other reasons include the following:

- Developers writing for the desktop are not motivated to spend time tuning the sizes of scalar objects. (The fact that any storage saving from using a smaller type is insignificant and the instruction timings for the different types is often very similar if not identical is not the issue, because many developers believe there are performance differences.) There are often few resources (e.g., developers time, performance-monitoring tools) provided by management to motivate developers to think about performance issues in this environment.
- Processors used for embedded applications usually have instructions specifically designed to efficiently handle 8-bit data. Programs often have storage constraint when executing in such environments. There are real benefits to be had from being able to use an 8- or 16-bit data type. The technical case, showing that performance improvements are available, is visible to developers and their managers. The demands of the application can also require management to make the resources available to consider performance as the application is being designed, coded, and maintained.

It might be claimed that embedded applications are oriented to 8-bit data. The application requirements control the choice of processor, not the other way around. Hardware cost is important (designers worry over pennies, which add up when millions of units are to be manufactured). Processors that handle 8-bit data are usually cheaper to manufacture and the interfacing costs are lower (fewer wires to the outside world).

---

standard unsigned integer

The type **_Bool** and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. 487

Commentary

This defines the term *standard unsigned integer type*.

usual arithmetic conversions _Bool 476 large enough to store 0 and 1

The unsignedness of the type **_Bool** is unlikely to be visible to the developer. The usual arithmetic conversions will always promote objects having this type to the type **int**.

C90

Support for the type **_Bool** is new in C99.

C++

In C++ the type **bool** is classified as an integer type (3.9.1p6). However, it is a type distinct from the signed and unsigned integer types.

Coding Guidelines

Although the type **_Bool** may, technically, be an integer type, there are benefits to treating it as a distinct type. However, there is a potential cost in doing so. Treating it as a distinct type reinforces the (technically incorrect) developer expectation that implementations will treat it as purely a boolean type, having one of two values. If a value other than 0 or 1 (through use of a construct exhibiting undefined behavior) is stored in such an object, implicit assumptions in a program may no longer hold. These coding guidelines assume that the benefits of treating the type **_Bool** as a distinct type significantly outweigh the potential costs The fact that the type **_Bool** is classified as an unsigned integer type is not of any practical significance to coding guidelines.

boolean role 476

---

extended unsigned integer

The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. 488

Commentary

This defines the term *extended unsigned integer type*. By definition any unsigned integer type that does not have a corresponding extended signed integer type is not as an *extended unsigned integer type*.

Common Implementations

The concept of extended integer types is still too new to be able to say anything about common implementations. The Motorola AltiVec implementation[22] supports the type **pixel**. This uses 16 bits to represent a display pixel (a 1/5/5/5 bit interpretation of the bits is used, and the type is treated like an **unsigned short**).

Coding Guidelines

The use of extended unsigned integer types violates the guideline recommendation dealing with using extensions and using a single integer type. Such types are not usually provided by implementations. Support for them, by an implementation, implies customer demand. Given such demand it is likely that they will be used, in some cases, by developers. Following coding guidelines is unlikely to take precedence in these cases, and nothing more is said about the issue.

?? extensions
cost/benefit
480.1 object
int type only

---

489 The standard and extended unsigned integer types are collectively called *unsigned integer types*.[30)]

unsigned integer types

Commentary

This defines the term *unsigned integer types*. It is common practice to use the term *unsigned types* to denote these types.

483 signed integer types

C90

Explicitly including the extended signed integer types in the definition is new in C99.

---

490 28) Implementation-defined keywords shall have the form of an identifier reserved for any use as described in 7.1.3.

footnote 28

Commentary

An implementation-defined keyword is an extension, however it is spelled. In specifying how new keywords are to be added to the language the Committee is recognizing that this is often done by translator vendors. By listing the identifier spellings that should be used, the Committee is also giving a warning to developers not to use them in their own declarations.

The incentive for vendors to use these spellings is that they can claim their extensions have been added in a standards-conforming way. What do developers gain from it? Existing code is less likely to be broken (because it is not supposed to use these spellings) and the Committee continues to have the option of using nonreserved words in future language enhancements (the Committee never makes extensions).

C90

The C90 Standard did not go into this level of detail on implementation extensions.

C++

The C++ Standard does not say anything about how an implementation might go about adding additional keywords. However, it does list a set of names that is always reserved for the implementation (17.4.3.1.2).

Other Languages

Most languages say nothing about possible extensions to themselves, perhaps in the hope that there will not be any such extensions. Those that do rarely specify a list of reserved identifiers.

Common Implementations

Suggestions about the spelling of additional keywords was not given in C90 and vendors used a variety of identifiers, only a few of them following the usage described in 7.1.3. One of the most commonly seen keyword extensions, **`asm`**, is not in the list of reserved names. The keywords **`near`** and **`far`** (and sometimes **`tiny`** and **`huge`**) were used by translators targeting the Intel x86 processor when many of the applications were predominantly 16 bit. While most applications for this processor are now 32 bit, support for these keywords is kept for backwards compatibility. These keywords have also been adopted by translators targeting other processors, where pointers of various widths need to be supported.

The token sequence **`long long`** was a relatively common extension in C90 implementations. This usage did not introduce a new keyword; it used existing keywords to create a new type.

Experience shows that developers prefer keywords that are short and meaningful, and they often complain that having to prefix keywords with underscores is irksome. A solution used by some implementations that have added extra keywords is to define matching macro names (e.g., `gcc` defines the keywords

typing minimization

`__attribute__` and `__typeof__`; It also predefines the macro names `attribute` and `typedef`, which expand to the respective keywords).

The Keil compiler[19] supported the keyword **`bit`**, for some processors, well before the C99 Standard was published.

---

footnote 29

29) Therefore, any statement in this Standard about signed integer types also applies to the extended signed integer types. 491

Commentary

Because the extended integer types are included in the definition of signed integer types, any statement that refers to the term *signed integer types* also includes any extended integer types.

C++

The C++ Standard does not place any requirement on extended integer types that may be provided by an implementation.

Common Implementations

It remains to be seen whether implementations abide by this non-normative footnote.

Coding Guidelines

Any guideline recommendation that applies to the signed integer types also applies to any extended signed integer types. The additional guidelines that apply to extended signed integer types are those that apply generally to the use of extensions. The additional complexity introduced into the usual arithmetic conversions needs to be considered.

implementation extensions
usual arithmetic conversions

---

footnote 30

30) Therefore, any statement in this Standard about unsigned integer types also applies to the extended unsigned integer types. 492

Commentary

footnote 29 491

The discussion on statements that apply to the signed integer types is also applicable here.

---

standard integer types
extended integer types

The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*, the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*. 493

Commentary

This defines the terms *standard integer types* and *extended integer types*. Note that this definition does not include the type **`char`** or the enumerated types. These are included in the definition of the term *integer types*.

integer types 519

C++

3.9.1p7

*Types **`bool`**, **`char`**, **`wchar_t`**, and the signed and unsigned integer types are collectively called integral types.[43)] A synonym for integral type is integer type.*

Footnote 43

*43) Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to **`int`**, **`unsigned int`**, **`long`**, or **`unsigned long`**, as specified in 4.5.*

enumeration constant type

The issue of enumerations being distinct types, rather than integer types, is discussed elsewhere.

---

integer types
relative ranges
rank
relative ranges

For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type. 494

Commentary

integer types
sizes
conversion rank

The specification of the sizes of integer types provides a minimum range of values that each type must be able to represent. Nothing is said about maximum values. The specification of rank is based on the names of the types, not their representable ranges. This requirement prevents, for instance, an implementation from supporting a greater range of representable values in an object of type **short** than in an object of type **int**. An implementation could choose to support an integer type, with a given conversion rank, capable of representing the same range of values as an integer type with greater rank. But an integer type cannot be capable of representing a greater range of values than another integer type (of the same sign) without also having a greater rank.

conversion rank

The concept of rank is new in C99.

C90

There was no requirement in C90 that the values representable in an unsigned integer type be a subrange of the values representable in unsigned integer types of greater rank. For instance, a C90 implementation could make the following choices:

```
1  SHRT_MAX  ==   32767  /* 15 bits */
2  USHRT_MAX ==  262143  /* 18 bits */
3  INT_MAX   ==   65535  /* 16 bits */
4  UINT_MAX  ==  131071  /* 17 bits */
```

No C90 implementation known to your author fails to meet the stricter C99 requirement.

C++

3.9.1p2

In this list, each type provides at least as much storage as those preceding it in the list.

C++ appears to have a lower-level view than C on integer types, defining them in terms of storage allocated, rather than the values they can represent. Some deduction is needed to show that the C requirement on values also holds true for C++:

3.9.1p3

*For each of the signed integer types, there exists a corresponding (but different) unsigned integer type: "**unsigned char**", "**unsigned short int**", "**unsigned int**", and "**unsigned long int**," each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type[40] ;*

They occupy the same storage,

3.9.1p3

that is, each signed integer type has the same object representation as its corresponding unsigned integer type. The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.

they have a common set of values, and

3.9.1p4

*Unsigned integers, declared **unsigned**, shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.[41]*

more values can be represented as the amount of storage increases.

QED.

Coding Guidelines

Each integer type's ability to represent values, relative to other integer types, is something developers learn early and have little trouble with thereafter. It is developers' expectation of specific integer types being able to represent the same range of values as other integer types that is often the root cause of faults and portability problems.

In a 16-bit environment there is often an expectation that the type **int** is represented in 16 bits, a width that differs from the type **long** and (sometimes) pointer types. In a 32-bit environment there is often an expectation that the pointer types and the types **int** and **long** are represented using the same amount of storage, each being capable of storing any value that can be represented in the other's type.

widest type assumption

The introduction of the type **long long** in C99 uncovered a new expectation— that the type **long** is the widest integer type (and the less reasonable expectation that the type **long** occupies at least the same amount of storage as a pointer). The typedef `intmax_t` was introduced to provide a name for the concept of widest integer type to prevent this issue from causing a problem in the future. However, this typedef name does not solve the problem in existing code.

---

positive signed integer type subrange of equivalent unsigned type

The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[31)] 495

Commentary

This is a requirement on the implementation (even though it does not use the term *shall*). The standard is doing more than enshrining various, low-level representation issues within the language. If the type of operands differ in their sign only, the result of the usual arithmetic conversions is an unsigned type. This requirement ensures that such a conversion does not lead to any surprises if the operand with signed type is positive. This requirement does rule out implementations using the following:

usual arithmetic conversions

- More representation bits in the signed type; for instance, the signed type using 24 bits and the unsigned type 16 bits.
- Combinations of integer representations; for instance, two's complement for a particular signed type and BCD for its corresponding unsigned type (although the use of BCD is ruled out by other requirements).

sign bit representation

However, it does not rule out the possibility of an unsigned integer type being able to represent significantly more values than its corresponding signed type.

object representation same padding signed/unsigned

Other Languages

Most languages do not specify representation details down to this level.

Common Implementations

Some host processors contain instructions for operating on BCD representations (e.g.,the Intel x86). A few older processors aimed at the business language (Cobol) market had no instructions for performing arithmetic on binary integer representations (e.g., ICL System 25). C on such hardware would be problematic.

---

unsigned computation modulo reduced

A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type. 496

Commentary

The common term for such operations is that they *wrap*.

unsigned integer conversion to

This specification describes the behavior of arithmetic operations for the vast majority of existing processors operating on values having unsigned types. Modulo arithmetic is a long-established part of mathematics (number theory). However, mathematicians working in the field of program correctness often frown on reliance on modulo arithmetic operations in programs (they represent a discontinuity). The **%** operator

% operator result

provides another mechanism for obtaining modulo behavior.

The LIA (Language Independent Arithmetic) standards[17] treat both nonrepresentable signed and unsigned values in the same way, a mathematical view of the world where values are never intended to exceed their maximum values. C's view is based on how existing processors operate on unsigned quantities.

Other Languages

Languages that support an unsigned integer type usually specify a similar behavior.

Common Implementations

This behavior describes what most processor operations on unsigned values already do. In those rare cases where a processor does not support unsigned operations, calls to internal library functions have to be made.

Coding Guidelines

Are the people working in the field of proving program correctness themselves correct to recommend against the use of modulo arithmetic? It seems to your author that such people are more interested in making programs fit their favorite mathematical theories and algorithms than making the mathematics fit the programs that commercial developers write. In practice the truncation of significant bits (which is how the engineers who originally designed the processor operations thought about it) for operations on unsigned values does not differ from the behavior commonly seen for signed values. However, the standard specifies a well-defined behavior in the former case and undefined behavior in the latter case. However, the difference is not in the commonly seen behaviors, or the specification provided in the standard. The difference is in the commonly seen developer intent.

arithmetic operation
exceptional condition
exception condition

Developers sometimes write code that relies on the modulo behavior of operations on unsigned values. It is very rare for developers to write code that relies on the common processor behavior on signed value overflow. It is not possible to imply from an object having an unsigned type that developers intend operations using it to wrap. The fact that modulo arithmetic is defined by the standard and used by developers some of the time does not mean that this behavior is always intended.

Are there any common cases? Are objects usually defined with unsigned integer types purely to make use of the larger range of values available, or is the modulo behavior on overflow intended? A similar question can be asked of a cast to an unsigned integer type.

It is your author's experience (who has no figures to back his view up) that most operations whose mathematical result is not within the range of the type are not intended by developers. Wanting the result to be modulo-reduced is the less common case. For this reason (and the effects of the mathematical puritans) many coding guideline documents specify that operations on operands having unsigned type should not generate a value that is different from its modulo-reduced value (i.e., operations on unsigned values are not allowed to *overflow*).

Some algorithms depend on modulo arithmetic behavior (e.g., in cryptography). Recommending against all such usage is not constructive. Documenting all such dependencies is a useful aid to readers because they can then assume that in all other cases no such behavior is intended.

Rev 496.1 Those operations involving operands, having an unsigned integer type, where it is known that the result may need to be modulo-reduced shall be commented as such.

---

497 There are three *real floating types*, designated as **float**, **double**, and **long double**.[32)]

floating types three real

Commentary

This defines the term *real floating types*. The possibility of additional floating-point types is listed in future language directions

floating types future language directions

The organization of the floating types has a similar structure to that commonly seen in the handling of the integer types **short**, **int**, and **long**. The type **double** is often thought of in terms of the floating-point

480 standard signed integer types

equivalent of **int**. On some implementations it has the same size as the type **float**, on other implementations it has the same size as the type **long double**, and on a few implementations its size lies between these two types. One difference between integer and floating types is that in the latter case an implementation is given much greater freedom in how operations on operands having these types are handled. The header **<math.h>** defines the typedefs `float_t` and `double_t` for developers who want to define objects having types that correspond to how an implementation performs operations.

FLT_EVAL_METHOD

base document

The type **long double** was introduced in C90. It was not in K&R C.

C90

What the C90 Standard calls *floating types* includes complex types in C99. The term *real floating types* is new in C99.

Other Languages

Many languages only support a single floating-point type. Some implementations of these languages, for instance Fortran, include extensions that allow the size of the floating-point type to be specified. The Fortran 90 standard provides a mechanism for developers to specify decimal precision and exponent range. The Ada Standard gives permission for an implementation to support the optional predefined types **SHORT_FLOAT** and **LONG_FLOAT**. Java has the types **float** and **double** type, but does not have a type **long double**.

Common Implementations

The original K&R specification treated the token sequence **long float** as a synonym for **double** (this support was removed during the early evolution of C[25] although some implementations continue to support it[10]). However, some vendors continue to support this usage.[11]

When floating-point operations are performed using hardware, there are usually two or more different floating-point formats (number of bits). When these operations are carried out in software, sometimes only a single representation is available (in some cases the vendor only provides floating-point support to enable them to claim conformance to the C Standard). The IEC 60559 Standard defines single- and double-precision formats. It also supports an extended precision form for both of these representations.

IEC 60559

The Cray processors are unusual in that the types **float** and **double** are both represented in the same number of bits. It is the size of **float** that has been increased to 64 bits, not **double** reduced to 32 bits. The IBM ILE C compiler[13] supports a packed decimal data type. The declaration `decimal(6, 2) x;` declares x to have six decimal digits before the decimal point and two after it.

Coding Guidelines

The simple approach of using as much accuracy as possible, declaring all floating-point objects as type **long double**, does not guarantee that algorithms will be well behaved. There is no substitute for careful thought and this is even more important when dealing with floating-point representation.

The type **double** tends to be the floating-point type used by default (rather like the type **int**). Execution time performance is an issue that developers often think about when dealing with floating-point types, sometimes storage capacity (for large arrays) can also be an issue. The type **double** has traditionally been the recommended floating type, for developers to use by default, although in many cases the type **float** provides sufficient accuracy. Given the problems that many developers have in correctly using floating types, a more worthwhile choice of guideline recommendation might be to recommend against their use completely.

It may be possible to trade execution-time performance against accuracy of the final results, but this is not always the case. For instance, some processors perform all floating-point operations to the same accuracy and the operations needed to convert between types (less/more accurate) can decrease execution performance. For processors that operate on formats of different sizes, it is likely that operations on the smaller size will be faster. The question is then whether enough is understood, by the developer, about the algorithm to know if the use a floating-point type with less accuracy will deliver acceptable results.

floating-point architectures

In practice few algorithms, let alone applications, require the stored precision available in the types **double** or **long double**. However, a minimum amount of accuracy may be required in the intermediate result of expression evaluation. In some cases the additional range supported by the exponents used in wider types is

FLT_EVAL_METHOD

required by an application. exponent

Given the degree of uncertainty about the costs and benefits in using any floating types, this coding guideline subsection does not make any recommendations.

Table 497.1: Occurrence of floating types in various declaration contexts (as a percentage of all floating types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	`typedef`	Member	Total
`float`	35.2	15.1	8.3	0.7	21.0	80.3
`double`	8.5	7.9	0.5	0.7	2.2	19.7
`long double`	0.0	0.0	0.0	0.0	0.0	0.0
Total	43.6	22.9	8.8	1.5	23.2	

498 The set of values of the type **`float`** is a subset of the set of values of the type **`double`**;

Commentary

This is a requirement on the implementation. There are no special values that can be represented in the type **`float`** that cannot also be represented in the type **`double`**.

Common Implementations

In IEC 60559 the significand, in double-precision, contains more representation bits than in single-precision (assuming these representations are chosen for the types **`float`** and **`double`**). Similarly, the exponent can represent a greater range of powers of 2.

Coding Guidelines

With 30 additional bits of significand in IEC 60559, there is a 1 in 10^9 chance of a value represented in double-precision having an exact value presentation in single-precision (leaving aside the possible extra exponent range available in the type **`double`**).

499 the set of values of the type **`double`** is a subset of the set of values of the type **`long double`**. double values subset of long double

Commentary

This is a requirement on the implementation. There are no special values that can be represented in the type **`double`** that cannot also be represented in the type **`long double`**.

Common Implementations

On an Intel x86-based host the type **`double`** is usually 64 total bits. Some implementations also choose this representation for the type **`long double`**. Others make use of the extended precision mode supported by the floating-point unit, which is used for performing all floating-point operations. In these cases the type **`long double`** is often represented by 80 or 96 bits.

For 128-bit **`long double`** most IEC 60559 implementations use the format of 1–15–113 bits for sign-exponent-significand (and a hidden bit just like single and double).

Some implementations use two contiguous **`double`**s to represent the type **`long double`**. This format has problems near `DBL_MIN` in that no extra precision, than available in the type **`double`**, is available. Also, while it can represent `2*DBL_MAX`, there is normally no additional range available. There is also the issue of the interpretation of `LDBL_EPSILON`. floating types characteristics *_EPSILON

500 There are three *complex types*, designated as **`float _Complex`**, **`double _Complex`**, and **`long double _Complex`**.[33] complex types

Commentary

The introduction of complex types in C99 was based on a marketing decision. The Committee wanted to capture the numerical community, which was continuing to use Fortran. The Committee was told that one

of the major reasons engineers and scientists prefer Fortran is that it does a better job of supporting their requirements for numerical computation, support for complex types being one of these requirements. Fortran does not support complex integer types and there did not seem to be sufficient utility to introduce such types in C99.

A new header, **<complex.h>**, has been defined for performing operations on objects having the type **_Complex**. Annex G (informative) specifies IEC 60559-compatible complex arithmetic.

C90

Support for complex types is new in C99.

C++

In C++ `complex` is a template class. Three specializations are defined, corresponding to each of the floating-point types.

26.2p1 *The header* **<complex>** *defines a template class, and numerous functions for representing and manipulating complex numbers.*

26.2p2 *The effect of instantiating the template* `complex` *for any type other than float, double, or long double is unspecified.*

The C++ syntax for declaring objects of type **complex** is different from C.

```
#ifdef __cplusplus

#include <complex>

typedef complex<float> float_complex;
typedef complex<double> double_complex;
typedef complex<long double> long_double_complex;

#else

#include <complex.h>

typedef float complex float_complex;
typedef double complex double_complex;
typedef long double complex long_double_complex;
#endif
```

Other Languages

Fortran has contained complex types since an early version of that standard. Few other languages specify a built-in complex type (e.g., Ada, Common Lisp, and Scheme).

Common Implementations

Very few processors support instructions that operate on complex types. Implementations invariably break down the operations into their constituent real and imaginary parts, and operate on those separately.

`gcc` supports integer complex types. Any of the integer type specifiers may be used.

Coding Guidelines

The use of built-in language types may offer some advantages over developer-defined representations (optimizers have more information available to them and may be able to generate more efficient code). However, the cost involved in changing existing code to use this type is likely to be larger than the benefits reaped.

501 The real floating and complex types are collectively called the *floating types*. floating types

Commentary

This defines the term *floating types*.

C90

What the C90 Standard calls *floating types* includes complex types in C99.

Coding Guidelines

There is plenty of opportunity for confusion over this terminology. Common developer usage did not use to distinguish between complex and real types; it did not need to. Developers who occasionally make use of floating-point types will probably be unaware of the distinction, made by the C Standard between real and complex. The extent to which correct terminology will be used within the community that uses complex types is unknown.

502 For each floating type there is a *corresponding real type*, which is always a real floating type.

Commentary

This defines the term *corresponding real type*. The standard does not permit an implementation to support a complex type that does not have a matching real type. Given that a complex type is composed of two real components, this may seem self-evident. However, this specification prohibits an implementation-supplied complex integer type.

503 For real floating types, it is the same type.

Commentary

It is the same type in that the same type specifier is used in both the real and complex declarations.

504 For complex types, it is the type given by deleting the keyword **_Complex** from the type name.

Commentary

The keyword **_Complex** cannot occur as the only type specifier, because it has no implicit real type. One of the real type specifiers has to be given.

C++

In C++ the complex type is a template class and declarations involving it also include a floating-point type bracketed between < > tokens. This is the type referred to in the C99 wording.

505 Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; complex type representation

Commentary

This level of specification ensures that C objects, having a complex type, are likely to have the same representation as objects of the same type in Fortran within the same host environment. Such shared representations simplifies the job of providing an interface to library functions written in either language.

Rationale

The underlying implementation of the complex types is Cartesian, rather than polar, for overall efficiency and consistency with other programming languages. The implementation is explicitly stated so that characteristics and behaviors can be defined simply and unambiguously.

C++

The C++ Standard defines `complex` as a template class. There are no requirements on an implementation's representation of the underlying values.

Other Languages

Other languages that contain complex as a predefined type do not usually specify how the components are represented in storage. Fortran specifies that the type **`complex`** is represented as an ordered pair of real data.

Common Implementations

Some processors have instructions capable of loading and storing multiple registers. Such instructions usually require adjacent registers (based on how registers are named or numbered). Requiring adjacent registers can significantly complicate register allocation and an implementation may chose not to make use of these instructions.

Coding Guidelines

This requirement does more than imply that the **`sizeof`** operator applied to a complex type will return a value that is exactly twice that returned when the operator is applied to the corresponding real type. It exposes other implementation details that developers might want to make use of. The issues involved are discussed in the following sentence.

Example

```
1  #include <stdlib.h>
2
3  double _Complex *f(void)
4  {
5  /*
6   * Not allocating an even number of doubles.  Suspicious?
7   */
8  return (double _Complex *)malloc(sizeof(double) * 3);
9  }
```

---

complex component representation

the first element is equal to the real part, and the second element to the imaginary part, of the complex number. 506

Commentary

This specification lists additional implementation details that correspond to the Fortran specification. This requirement means that complex types are implemented using Cartesian coordinates rather than polar coordinates. A consequence of the choice of Cartesian coordinates (rather that polar coordinates) is that there are four ways of representing 0 and eight ways of representing infinity (where n represents some value):

`+0 + i*0`	`-0 + i*0`	`-0 - i*0`	`+0 - i*0`
$+\infty + i*n$	$+\infty + i*\infty$	$n + i*\infty$	$-\infty + i*\infty$
$-\infty + i*n$	$-\infty - i*\infty$	$n - i*\infty$	$+\infty - i*\infty$

The library functions `creal` and `cimag` provide direct access to these components.

C++

Clause 26.2.3 lists the first parameter of the complex constructor as the real part and the second parameter as the imaginary part. But, this does not imply anything about the internal representation used by an implementation.

Other Languages

Fortran specifies the Cartesian coordinate representation.

Coding Guidelines

The standard specifies the representation of a complex type as a two-element array of the corresponding real types. There is nothing implementation-defined about this representation and the guideline recommendation against the use of representation information is not applicable.

?? representation information using

One developer rationale for wanting to make use of representation information, in this case, is efficiency. Modifying a single part of an object with complex type invariably involves referencing the other part; for instance, the assignment:

```
1   val = 4.0 + I * cimag(val);
```

may be considered as too complicated for what is actually involved. A developer may be tempted to write:

```
1   *(double *)&val = 4.0;
```

as it appears to be more efficient. In some cases it may be more efficient. However, use of the address-of operator is likely to cause translators to be overly cautious, only performing a limited set of optimizations on expressions involving `val`. The result could be less efficient code. Also, the second form creates a dependency on the declared type of `val`. Until more experience is gained with the use of complex types in C, it is not possible to evaluate whether any guideline recommendation is worthwhile.

Example

It is not possible to use a typedef name to parameterize the kind of floating-point type used in the following function.

```
1   double f(double _Complex valu, _Bool first)
2   {
3   double *p_c = (double *)&valu;
4
5   if (first)
6      return *p_c;       /* Real part. */
7   else
8      return *(p_c + 1); /* Imaginary part. */
9   }
```

---

507 The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*.

basic types

Commentary

This defines the term *basic types* (only used in this paragraph and footnote 34) which was also defined in C90. The term *base types* is sometimes used by developers.

512 footnote 34

C++

The C++ Standard uses the term *basic types* three times, but never defines it:

> *[Note: even if the implementation defines two or more basic types to have the same value representation, they are nevertheless different types.]*

3.9.1p10

> *The identities among certain predefined operators applied to basic types (for example, ++a ≡ a+=1) need not hold for operator functions. Some predefined operators, such as +=, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.*

13.5p7

Footnote 174

174) An implicit exception to this rule are types described as synonyms for basic integral types, such as `size_t` *(18.1) and* `streamoff` *(27.4.1).*

Coding Guidelines

This terminology is not commonly used outside of the C Standard's Committee.

---

types different even if same representation

Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.[34)] 508

Commentary

The type checking rules are independent of representation (which can change between implementations). A type is a property in its own right that holds across all implementations. For example, even though the type **`char`** is defined to have the same range and representation as either of the types **`signed char`** or **`unsigned char`**, it is still a different type from them.

char 537 separate type

Other Languages

Some languages go even further and specify that all user defined types, even of scalars, are different types. These are commonly called strongly typed languages.

Common Implementations

Once any type compatibility requirements specified in the standard have been checked, implementations are free to handle types having the same representation in the same way. Deleting casts between types having the same representation is so obvious it hardly merits being called an optimization. Some optimizers use type information when performing alias analysis— for instance, in the following definition:

alias analysis

```
1  void f(int *p1, long *p2, int *p3)
2  { /* ... */ }
```

It might be assumed that the objects pointed to by `p1` and `p2` do not overlap because they are pointers to different types, while the objects pointed to by `p1` and `p3` could overlap because they are pointers to the same type.

Coding Guidelines

typedef is synonym

C does not provide any mechanism for developers to specify that two typedef names, defined using the same integer type, are different types. The benefits of such additional type-checking machinery are usually lost on the C community.

Example

```
1  typedef int APPLES;
2  typedef int ORANGES;
3
4  APPLES coxes;
5  ORANGES jafa;
6
7  APPLES totals(void)
8  {
9  return coxes + jafa; /* Adding apples to oranges is suspicious. */
10 }
```

---

footnote 31

31) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions. 509

Commentary

This interchangeability does not extend to being considered the same for common initial sequence purposes. The sentence that references this footnote does not discuss any alignment issues. This footnote is identical to footnote 39.

common initial sequence

565 footnote 39

Prior to C90 there were no function prototypes. Developers expected to be able to interchange arguments that had signed and unsigned versions of the same integer type. Having to cast an argument, if the parameter type in the function definition had a different signedness, was seen as counter to C's easy-going type-checking system and a little intrusive. The introduction of prototypes did not completely do away with the issue of interchangeability of arguments. The ellipsis notation specifies that nothing is known about the expected type of arguments.

ellipsis supplies no information

Similarly, for function return values, prior to C99 it was explicitly specified that if no function declaration was visible the translator provided one. These implicit declarations defaulted to a return type of **int**. If the actual function happened to return the type **unsigned int**, such a default declaration might have returned an unexpected result. A lot of developers had a casual attitude toward function declarations. The rest of us have to live with the consequences of the Committee not wanting to break all the source code they wrote. The interchangeability of function return values is now a moot point, because C99 requires that a function declaration be visible at the point of call (a default declaration is no longer provided).

Having slid further down the slippery slope, we arrive at union types. From the efficiency point of view, having to assign a member of a union to another member, having the corresponding (un)signed integer type, knowing that the value is representable, seems overly cautious. If the value is representable in both types, it is a big simplification not to have to be concerned about which member was last assigned to.

This footnote does not explicitly discuss casting pointers to the same signed/unsigned integer type. If objects of these types have the same representation and alignment requirements, which they do, and the value pointed at is within the range common to both types, everything ought to work. However, *meant to imply* does not explicitly apply in this case.

DR #070

The program is not strictly conforming. Since many pre-existing programs assume that objects with the same representation are interchangeable in these contexts, the C Standard encourages implementors to allow such code to work, but does not require it.

The program referred to, in this DR, was very similar to the following:

```
1  #include <stdio.h>
2
3  void output(c)
4  int c;
5  {
6  printf("C == %d\n", c);
7  }
8
9  void DR_070(void)
10 {
11 output(6);
12 /*
13  * The following call has undefined behavior.
14  */
15 output(6U);
16 }
```

Other Languages

Few languages support unsigned types as such. Languages in the Pascal family allow subranges to be specified, which could consist of nonnegative values only. However, such subrange types are not treated any differently by the language semantics than when the subrange includes negative values. Consequently, other

languages tend to say nothing about the interchangeability of objects having the corresponding signed and unsigned types.

Common Implementations

The standard does not require that this interchangeability be implemented. But it gives a strong hint to implementors to investigate the issue. There are no known implementations that don't do what they are *imply*ed to do.

Coding Guidelines

function ?? declaration use prototype

If the guideline recommendation dealing with use of function prototypes is followed, the visible prototype will cause arguments to be cast to the declared type of the parameter. The function return type will also always be known. However, for arguments corresponding to the ellipsis notation, translators will not perform any implicit conversions. If the promoted type of the argument is not compatible with the type that appears in any invocation of the `va_arg` macro corresponding to that argument, the behavior is undefined. Incompatibility between an argument type and its corresponding parameters type (when no prototype is visible) is known to be a source of faults (hence the guideline recommendation dealing with the use of prototypes). So it is to be expected that the same root cause will also result in use of the `va_arg` macro having the same kinds of fault. However, use of the `va_arg` macro is relatively uncommon and for this reason no guideline recommendation is made here.

function ?? declaration use prototype

union member when written to

Signed and unsigned versions of the same type may appear as members of union types. However, this footnote does not give any additional access permissions over those discussed elsewhere. Interchangeability of union members is rarely a good idea.

effective type

What about a pointer-to objects having different signed types? Accessing objects having different types, signed or otherwise, may cause undefined behavior and is discussed elsewhere. The interchangeability being discussed applies to values, not objects.

Example

```
1  union {
2          signed int m_1;
3          unsigned int m_2;
4         } glob;
5
6  extern int g(int, ...);
7
8  void f(void)
9  {
10 glob.m_2=3;
11 g(2, glob.m_1);
12 }
```

footnote 32

32) See "future language directions" (6.11.1). 510

footnote 33

33) A specification for imaginary types is in informative annex G. 511

Commentary

Normative references

This annex is informative, not normative, and is applicable to IEC 60559-compatible implementations.

C++

There is no such annex in the C++ Standard.

footnote 34

34) An implementation may define new keywords that provide alternative ways to designate a basic (or any other) type; 512

Commentary

Some restrictions on the form of an identifier used as a keyword are given elsewhere. A new keyword, provided by an implementation as an alternative way of designating one of the basic types, is not the same as a typedef name. Although a typedef name is a synonym for the underlying type, there are restrictions on how it can be used with other type specifiers (it also has a scope, which a keyword does not have). For instance, a vendor may supply implementations for a range of processors and chose to support the keyword **__int_32**. On some processors this keyword is an alternative representation for the type **long**, on others an alternative for the type **int**, while on others it may not be an alternative for any of the basic types.

490 footnote 28
typedef is synonym
type specifier syntax

C90

Defining new keywords that provide alternative ways of designating basic types was not discussed in the C90 Standard.

C++

The object-oriented constructs supported by C++ removes most of the need for implementations to use additional keywords to designate basic (or any other) types

Other Languages

Most languages do not give explicit permission for new keywords to be added to them.

Common Implementations

Microsoft C supports the keyword **__int64**, which specifies the same type as **long long**.

Coding Guidelines

Another difference between an implementation-supplied alternative designation and a developer-defined typedef name is that one is under the control of the vendor and the other is under the control of the developer. For instance, if `__int_32` had been defined as a typedef name by the developer, then it would be the developer's responsibility to ensure that it has the appropriate definition in each environment. As an implementation-supplied keyword, the properties of **__int_32** will be selected for each environment by the vendor.

The intent behind supporting new keywords that provide alternative ways to designate a basic type is to provide a mechanism for controlling the use of different types. In the case of integer types the guideline recommendation dealing with the use of a single integer type, through the use of a specific keyword, is applicable here.

480.1 object int type only

Example

```
/*
 * Assume vend_int is a new keyword denoting an alternative
 * way of designating the basic type int.
 */
typedef int DEV_INT;

unsigned DEV_INT glob_1;  /* Syntax violation. */
unsigned vend_int glob_2; /* Can combine with other type specifiers. */
```

---

513 this does not violate the requirement that all basic types be different.

Commentary

The implementation-defined keyword is simply an alternative representation, like trigraphs are an alternative representation of some characters.

---

514 Implementation-defined keywords shall have the form of an identifier reserved for any use as described in 7.1.3.

Commentary

footnote 28 490 This sentence duplicates the wording in footnote 28.

---

character types

The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*. 515

Commentary

This defines the term *character types*.

C++

Clause 3.9.1p1 does not explicitly define the term *character types*, but the wording implies the same definition as C.

Other Languages

Many languages have a character type. Few languages have more than one such type (because they do not usually support unsigned types).

Coding Guidelines

This terminology is not commonly used by developers who sometimes refer to *char types* (plural), a usage that could be interpreted to mean the type **char**. The term *character type* is not immune from misinterpretation either (as also referring to the type **char**). While it does have the advantage of technical correctness, there is no evidence that there is any cost/benefit in attempting to change existing, sloppy, usage.

Table 515.1: Occurrence of character types in various declaration contexts (as a percentage of all character types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	**typedef**	Member	Total
char	16.4	3.6	1.2	0.1	6.6	28.0
signed char	0.2	0.3	0.0	0.1	0.3	1.0
unsigned char	18.1	10.6	0.4	0.8	41.2	71.1
Total	34.7	14.6	1.5	1.0	48.2	

---

char range, representation and behavior

The implementation shall define **char** to have the same range, representation, and behavior as either **signed char** or **unsigned char**.[35) 516

Commentary

This is a requirement on the implementation. However, it does not alter the fact that the type **char** is a different type than **signed char** or **unsigned char**.

C90

This sentence did not appear in the C90 Standard. Its intent had to be implied from wording elsewhere in that standard.

C++

3.9.1p1 *A **char**, a **signed char**, and an **unsigned char** occupy the same amount of storage and have the same alignment requirements (3.9); that is, they have the same object representation.*

. . .

*In any particular implementation, a plain **char** object can take on either the same values as **signed char** or an **unsigned char**; which one is implementation-defined.*

In C++ the type **char** can cause different behavior than if either of the types **signed char** or **unsigned char** were used. For instance, an overloaded function might be defined to take each of the three distinct character types. The type of the argument in an invocation will then control which function is invoked. This is not an issue for C code being translated by a C++ translator, because it will not contain overloaded functions.

517 An *enumeration* comprises a set of named integer constant values.

enumeration
set of named
constants

Commentary

There is no phase of translation where the names are replaced by their corresponding integer constant. Enumerations in C are tied rather closely to their constant values. The language has never made the final jump to treating such names as being simply that— an abstraction for a list of names.

Rationale

The C89 Committee considered several alternatives for enumeration types in C:

1. leave them out;
2. include them as definitions of integer constants;
3. include them in the weakly typed form of the UNIX C compiler;
4. include them with strong typing as in Pascal.

The C89 Committee adopted the second alternative on the grounds that this approach most clearly reflects common practice. Doing away with enumerations altogether would invalidate a fair amount of existing code; stronger typing than integer creates problems, for example, with arrays indexed by enumerations.

Enumeration types were first specified in a document listing extensions made to the base document.

base document

Other Languages

Enumerations in the Pascal language family are distinct from the integer types. In these languages, enumerations are treated as symbolic names, not integer values (although there is usually a mechanism for getting at the underlying representation value). Pascal does not even allow an explicit value to be given for the enumeration names; they are assigned by the implementation. Java did not offer support for enumerated types until version 1.5 of its specification.

symbolic name

Coding Guidelines

The benefits of using a name rather than a number in the visible source to denote some property, state, or attribute is discussed elsewhere. Enumerated types provide a mechanism for calling attention to the association between a list (they may also be considered as forming a set) of identifiers. This association is a developer-oriented one. From the translators point of view there is no such association (unlike many other languages, which treat members as belonging to their own unique type). The following discussion concentrates on the developer-oriented implications of having a list of identifiers defined together within the same enumeration definition.

symbolic name

While other languages might require stronger typing checks on the use of enumeration constants and objects defined using an enumerated type, there are no such requirements in C. Their usage can be freely intermixed, with values having other integer types, without a diagnostic being required to be generated. Enumerated types were not specified in K&R C and a developer culture of using macros has evolved. Because enumerated types were not seen to offer any additional functionality, in particular no additional translator checking, that macros did not already provide, they have not achieved widespread usage.

Some coding guideline documents recommend the use of enumerated types over macro names because of the motivation that "using of the preprocessor is poor practice".[20] Other guideline documents specify ways of indicating that a sequence of macro definitions are associated with each other (by, for instance, using comments at the start and end of the list of definitions). The difference between such macro definition usage and enumerations is that the latter has an explicit syntax associated with it, as well as established practices from other languages.

The advantage of using enumerated types, rather than macro definitions, is that there is an agreed-on notation for specifying the association between the identifiers. Static analysis tools can (and do) use this

information to perform a number of consistency checks on the occurrence of enumeration constants and objects having an enumerated type in expressions. Without tool support, it might be claimed that there is no practical difference between the use of enumerated types and macro names. Tools effectively enforce stricter type compatibility requirements based on the belief that the definition of identifiers in enumerations can be taken as a statement of intent. The identifiers and objects having a particular enumerated type are being treated as a separate type that is not intended to be mixed with literals or objects having other types.

It is not known whether defining a list of identifiers in an enumeration type rather than as a macro definition affects developer memory performance (e.g., whether developers more readily recall them, their associated properties, or fellow group member names with fewer errors). The issue of identifier naming conventions based on the language construct used to define them is discussed elsewhere

identifier learning a list of source code context identifier

The selection of which, if any, identifiers should be defined as part of the same enumeration is based on concepts that exist within an application (or at least within a program implementing it), or on usage patterns of these concepts within the source code. There are a number of different methods that might be used to measure the extent to which the concepts denoted by two identifiers are similar. The human-related methods of similarity measuring, and mathematical methods based on concept analysis, are discussed elsewhere. Resnick[24] describes a measure of semantic similarity based on the *is-a* taxonomy that is based on the idea of shared information content.

categorization concept analysis

While two or more identifiers may share a common set of attributes, it does not necessarily mean that they should, or can, be members of the same enumerated type. The C Standard places several restrictions on what can be defined within an enumerated type, including:

- The same identifier, in a given scope, can only belong to one enumeration (Ada allows the same identifier to belong to more than one enumeration in the same scope; rules are defined for resolving the uses of such overloaded identifiers).
- The value of an enumeration constant must be representable in the type **int** (identifiers that denote floating-point values or string literals have to be defined as macro names).
- The values of an enumeration must be translation-time constants.

enumeration constant representable in int

Given the premise that enumerated types have an interpretation for developers that is separate from the C type compatibility rules, the kinds of operations supported by this interpretation need to be considered. For instance, what are the rules governing the mixing of enumeration constants and integer literals in an expression? If the identifiers defined in an enumeration are treated as symbolic names, then the operators applicable to them are assignment (being passed as an argument has the same semantics); the equality operators; and, perhaps, the relational operators, if the order of definition has meaning within the concept embodied by the names (e.g, the baud rates that follow are ordered in increasing speed).

The following two examples illustrate how symbolic names might be used by developers (they are derived from the clause on device- and class-specific functions in the POSIX Standard[16]). They both deal with the attributes of a serial device.

- A serial device will have a single data-transfer rate (for simplicity, the possibility that the input rate may be different from the output rate is ignored) associated with it (e.g., its baud rate). The different rates might be denoted using the following definition:

```
1  enum baud_rates {B_0, B_50, B_300, B_1200, B_9600, B_38400};
```

  where the enumerated constants have been ordered by data-transfer rate (enabling a test using the relational operators to return meaningful information).

- The following definition denotes various attributes commonly found in serial devices:

```
enum termios_c_iflag {
              BRKINT, /* Signal interrupt on break */
              ICRNL,  /* Map CR to NL on input     */
              IGNBRK, /* ignore break condition    */
              IGNCR,  /* Ignore CR                 */
              IGNPAR, /* Ignore characters with parity errors */
              INLCR,  /* Map NL to CR on input     */
              INPCK,  /* Enable input parity check */
              ISTRIP, /* Strip character           */
              IXOFF,  /* Enable start/stop input control  */
              IXON,   /* Enable start/stop output control */
              PARMRK  /* Mark parity errors               */
};
```

where it is possible that more than one of them can apply to the same device at the same time. These enumeration constants are members of a set. Given the representation of enumerations as integer constants, the obvious implementation technique is to use disjoint bit-patterns as the value of each identifier in the enumeration (POSIX requires that the enumeration constants in `termios_c_iflag` have values that are bitwise distinct, which is not met in the preceding definition). The bitwise operators might then be used to manipulate objects containing these values.

The order in which enumeration constants are defined in an enumerated type has a number of consequences, including:

- If developers recognize the principle used to order the identifiers, they can use it to aid recall.
- The extent to which relational operators may be applied.
- Enhancements to the code need to ensure that any ordering is maintained when new members are added (e.g., if a new baud rate, say 4,800, is introduced, should `B_4800` be added between `B_1200` and `B_9600` or at the end of the list?).

The extent to which a meaningful ordering exists (in the sense that subsequent readers of the source would be capable of deducing, or predicting, the order of the identifiers given a description in an associated comment) and can be maintained when applications are enhanced is an issue that can only be decided by the author of the code.

Rev 517.1 When a set of identifiers are used to denote some application domain attribute using an integer constant representation, the possibility of them belonging to an enumeration type shall be considered.

Cg 517.2 The value of an enumeration constant shall be treated as representation information.

Cg 517.3 If either operand of a binary operator has an enumerated type, the other operand shall be declared using the same enumerated type or be an enumeration constant that is part of the definition of that type.

If an enumerated type is to be used to represent elements of a set, it is important that the values of all of its enumeration constants be disjoint. Adding or removing one member should not affect the presence of any other member.

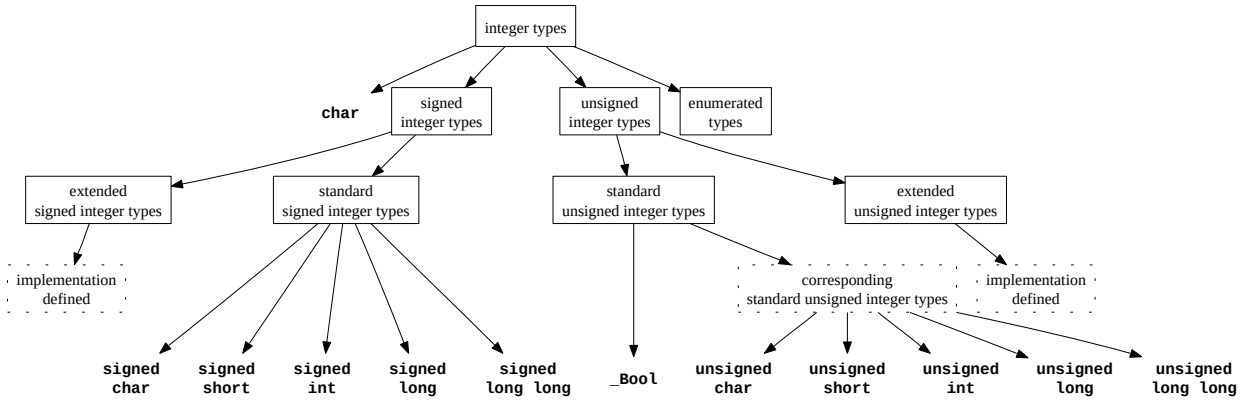

Figure 519.1: The integer types.

Usage

macro object-like

A study by Gravley and Lakhotia[9] looked at ways of automatically deducing which identifiers, defined as object-like macros denoting an integer constant, could be members of the same enumerated type. The heuristics used to group identifiers were based either on visual clues (block of **#define**s bracketed by comments or blank lines), or the value of the macro body (consecutive values in increasing or decreasing numeric sequence; bit sequences were not considered).

The 75 header files analyzed contained 1,225 macro definitions, of which 533 had integer constant bodies. The heuristics using visual clues managed to find around 55 groups (average size 8.9 members) having more than one member, the value based heuristic found 60 such groups (average size 6.7 members).

---

enumeration different type

Each distinct enumeration constitutes a different *enumerated type*. 518

Commentary

enumeration type compatible with compatible type if

Don't jump to conclusions. Each enumerated type is required to be compatible with some integer type. The C type compatibility rules do not always require two types to be the same. This means that objects declared to have an enumerated type effectively behave as if they were declared with the appropriate, compatible integer type.

C++

enumeration type compatible with

The C++ Standard also contains this sentence (3.9.2p1). But it does not contain the integer compatibility requirements that C contains. The consequences of this are discussed elsewhere.

Other Languages

Languages that contain enumerated types usually also treat them as different types that are not compatible with an integer type (even though this is the most common internal representation used by implementations).

Coding Guidelines

enumeration type compatible with

These coding guidelines maintain this specification of enumerations being different enumerated types and recommends that the requirement that they be compatible with some integer type be ignored.

---

integer types

The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*. 519

Commentary

This defines the term *integer types*. Some developers also use the terminology *integral types* as used in the C90 Standard.

C90

In the C90 Standard these types were called either *integral types* or *integer types*. DR #067 lead to these two terms being rationalized to a single term.

C++

> *Types* **bool**, **char**, **wchar_t**, *and the signed and unsigned integer types are collectively called integral types.*[43)] 3.9.1p7
> *A synonym for integral type is integer type.*

In C the type **_Bool** is an unsigned integer type and wchar_t is compatible with some integer type. In C++ they are distinct types (in overload resolution a **bool** or **wchar_t** will not match against their implementation-defined integer type, but against any definition that uses these named types in its parameter list).

In C++ the enumerated types are not integer types; they are a compound type, although they may be converted to some integer type in some contexts. 493 standard integer types

Other Languages

Many other languages also group the character, integer, boolean, and enumerated types into a single classification. Other terms used include *discrete types* and *ordinal types*.

Coding Guidelines

Both of the terms *integer types* and *integral types* are used by developers. Character and enumerated types are not always associated, in developers' minds with this type category.

Table 519.1: Occurrence of integer types in various declaration contexts (as a percentage of those all integer types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	**typedef**	Member	Total
char	1.8	0.4	0.1	0.0	0.7	3.1
signed char	0.0	0.0	0.0	0.0	0.0	0.1
unsigned char	2.0	1.2	0.0	0.1	4.6	7.9
short	0.7	0.3	0.0	0.0	0.4	1.4
unsigned short	2.3	0.8	0.1	0.1	3.2	6.5
int	28.4	10.6	4.2	0.1	6.4	49.7
unsigned int	5.6	3.6	0.3	0.1	4.2	13.8
long	3.0	1.2	0.1	0.1	0.8	5.1
unsigned long	4.8	1.9	0.2	0.1	2.1	9.1
enum	0.9	0.9	0.4	0.4	0.8	3.3
Total	49.6	20.8	5.4	0.9	23.2	

---

520 The integer and real floating types are collectively called *real types*. real types

Commentary

This defines the term *real types*.

C90

C90 did not include support for complex types and this definition is new in C99.

C++

The C++ Standard follows the C90 Standard in its definition of integer and floating types.

Coding Guidelines

This terminology is not commonly used outside of the C Standard. Are there likely to be any guideline recommendations that will apply to real types but not arithmetic types? If there are, then writers of coding guideline documents need to be careful in their use of terminology.

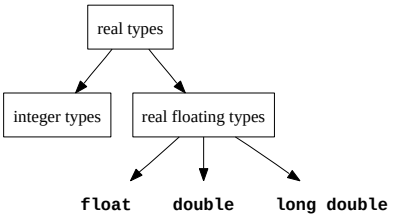

Figure 520.1: The real types.

---

arithmetic type

Integer and floating types are collectively called *arithmetic types*. 521

Commentary

This defines the term *arithmetic types*, so-called because they can appear as operands to the binary operators normally thought of as arithmetic operators.

C90

floating types 497
three real

Exactly the same wording appeared in the C90 Standard. Its meaning has changed in C99 because the introduction of complex types has changed the definition of the term floating types.

C++

The wording in 3.9.1p8 is similar (although the C++ complex type is not a basic type).
The meaning is different for the same reason given for C90.

Coding Guidelines

It is important to remember that pointer arithmetic in C is generally more commonly used than arithmetic on operands with floating-point types (see Table **??**, and Table **??**). There may be coding guidelines specific to integer types, or floating types, however, the category arithmetic type is not usually sufficiently general. Coding guidelines dealing with expressions need to deal with the general, type independent cases first, then the scalar type cases, and finally the more type specific cases.

scalar types 544

Writers of coding guideline documents need to be careful in their use of terminology here. C90 is likely to be continued to be used for several years and its definition of this term does not include the complex types.

---

type domain

Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types. 522

Commentary

This defines the terms *real type domain* and *complex type domain*. The concept of type domain comes from the mathematics of complex variables. Annex G describes the properties of the *imaginary type domain*. An

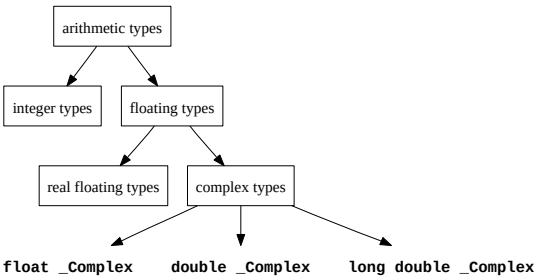

Figure 521.1: The arithmetic types.

implementation is not required to support this type domain. Many operations and functions return similar results in both the real and complex domains; for instance:

$$finite / ComplexInf \Rightarrow ComplexZero \quad (522.1)$$
$$finite * ComplexInf \Rightarrow ComplexInf \quad (522.2)$$

However, some operations and functions may behave differently in each domain; for instance:

$$\exp(Inf) \Rightarrow Inf \quad (522.3)$$
$$\exp(-Inf) \Rightarrow 0.0 \quad (522.4)$$
$$\exp(ComplexInf) \Rightarrow ComplexNaN \quad (522.5)$$

Both Inf and $-Inf$ can be viewed as the complex infinity under the Riemann sphere, making the result with an argument of complex infinity nonunique (it depends on the direction of approach to the infinity).

C90

Support for complex types and the concept of type domain is new in C99.

C++

In C++ `complex` is a class defined in one of the standard headers. It is treated like any other class. There is no concept of type domain in the C++ Standard.

Other Languages

While other languages containing a built-in complex type may not use this terminology, developers are likely to use it (because of its mathematical usage).

Coding Guidelines

Developers using complex types are likely to be familiar with the concept of *domain* from their mathematical education.

---

523 The **`void`** type comprises an empty set of values;

void is incomplete type

Commentary

Because types are defined in terms of values they can represent and operations that can be performed on them, the standard has to say what the **`void`** type can represent.

The **`void`** keyword plays many roles. It is the placeholder used to specify that a function returns no value, or that a function takes no parameters. It provides a means of explicitly throwing a value away (using a cast). It can also be used, in association with pointers, as a method of specifying that no information is known about the pointed-to object (pointer to a so-called *opaque* type).

The use of **`void`** in function return types and parameter definitions was made necessary because nothing appearing in these contexts had an implicit meaning— function returning **`int`** (not supported in C99) and function taking unknown parameters, respectively.

operator ()

C90

The **`void`** type was introduced by the C90 Committee. It was not defined by the base document.

base document

Other Languages

The keyword **`void`** is unique to C (and C++). Some other languages fill the role it plays (primarily in the creation of a generic pointer type) by specifying that no keyword appear. CHILL defines a different keyword, **`PTR`**, for use in this pointer role. Other languages that support a generic pointer type, or have special rules for handling pointers for recursive data structures use concepts that are similar to those that apply to the void type.

Coding Guidelines

generic pointer

The **void** type can be used to create an anonymous pointer type or a generic pointer type. The difference between these is intent. In one case there is the desire to hide information and in the other a desire to be able to accept any pointer type in a given context. It can be very difficult, when looking at source code, to tell the difference between these two uses.

coupling

Restricting access to implementation details (through information-hiding) is one way of reducing low-level coupling between different parts of a program. The authors of library functions (either third-party or project-specific) may want to offer a generalized interface to maximize the likelihood of meeting their users' needs without having to provide a different function for every type. Where the calling source code is known, is the use of pointers to **void** a lazy approach to passing information around or is it good design practice for future expansion? These issues are higher-level design issues that are outside of the scope of this book.

Usage

Information on keyword usage is given elsewhere (see Table 539.1, Table **??**, Table **??**, Table **??**, Table **??**, and Table **??**).

---

it is an incomplete type that cannot be completed. 524

Commentary

base document

The concept of an incomplete type was not defined in the base document, it was introduced in C90.

external linkage
exactly one external definition

Defining **void** to be an incomplete type removes the need for lots of special case wording in the standard. A developer defining an object to have the **void** type makes no sense (there are situations where it is of use to the implementation). But because it is an incomplete type, the wording that disallows objects having an incomplete type comes into play; there is no need to introduce extra wording to disallow objects being declared to have the **void** type. Being able to complete the **void** type would destroy the purpose of defining it to be incomplete in the first place.

---

derived type

Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows: 525

Commentary

This defines the term *derived types*. The rules for deciding whether two derived types are compatible are discussed in the clauses for those types.

translation limits
limit
type complexity

The translation limits clause places a minimum implementation limit on the complexity of a type and the number of external and block scope identifiers. However, there is no explicit limit on the number of types in a translation unit. Anonymous structure and union declarations, which don't declare any identifiers, in theory consume no memory; a translator can free up all the storage associated with them (but such issues are outside the scope of the standard).

C++

C++ has derived classes, but it does not define derived types as such. The term *compound types* fills a similar role:

3.9.2p1 *Compound types can be constructed in the following ways:*

Other Languages

Most languages allow some form of derived types to be built from the basic types predefined by the language. Not all languages support the range of possibilities available in C, while some languages define kinds of derived types not available in C— for instance, sets, tuples, and lists (as built-in types).

Common Implementations

The number of derived types is usually limited by the amount of storage available to the translator. In most cases this is likely to be large.

Coding Guidelines

The term *derived type* is not commonly used by developers. It only tends to crop up in technical discussions involving the C Standard by the Committee.

Derived types are not necessary for the implementation of any application; in theory, an integer type is sufficient. What derived types provide is a mechanism for more directly representing both how an application domain organizes its data and the data structures implied by algorithms (e.g., a linked list) used in implementing an application. Which derived types to define is usually a high-level design issue and is outside the scope of this book. Here we limit ourselves to pointing out constructions that have been known to cause problems in the past.

Table 525.1: Occurrence of derived types in various declaration contexts (as a percentage of all derived types appearing in all of these contexts, e.g., `int **ap[2]` is counted as two pointer types and one array type). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	`typedef`	Member	Total
`*`	30.4	37.6	3.1	0.8	5.6	77.5
array	3.3	0.0	4.4	0.0	3.0	10.8
`struct`	3.7	0.1	2.4	2.3	2.6	11.2
`union`	0.2	0.0	0.0	0.1	0.2	0.5
Total	37.7	37.8	10.0	3.3	11.3	

---

526 — An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*.[36)]

array type
array
contiguously allocated set of objects

Commentary

This defines the terms *array type* and *element type*.

Although array types can appear in declarations, they do not often appear as the types of operands. This is because an occurrence, in an expression context, of an object declared to have an array type is often converted into a pointer to its first element. Because of this conversion, arrays in C are often said to be second-class citizens (types). Note that the element type cannot be an incomplete or function type. The standard also specifies a lot of implementation details on how arrays are laid out in storage.

additive operators
pointer to object
array
row-major storage order

Other Languages

Nearly every language in existence has arrays in one form or another. Many languages treat arrays as having the properties listed here. A few languages simply treat them as a way of denoting a list of locations that may hold values (e.g., Awk and Perl allow the index expression to be a string); it is possible for each element to have a different type and the number of elements to change during program execution. A few languages restrict the element type to an arithmetic type— for instance, Fortran (prior to Fortran 90). The Java reference model does not require that array elements be contiguously allocated. In the case of multidimensional arrays there are implementation advantages to keeping each slice separate (in a garbage collected storage environment it keeps storage allocation requests small).

Coding Guidelines

The decision to use an array type rather than a structure type is usually based on answers to the following questions:

array type
when to use

- Is more than one element of the same type needed?
- Are the individual elements anonymous?
- Do all the elements represent the same applications-domain concept?
- Will individual elements be accessed as a sequence (e.g., indexed with a loop-control variable).

If the individual elements are not anonymous, they might better be represented as a structure type containing two members, for instance, the x and y coordinates of a location. The names of the members also provide a useful aid to remembering what is being represented. In those cases where array elements are used to denote different kinds of information, macros can be used to hide the implementation details. In the following an array holds color and height information. Using macros removes the need to remember which element of the array holds which kind of information. Using enumeration constants is another technique, but it requires the developer to remember that the information is held in an array (and also requires a greater number of modifications if the underlying representation changes):

```
#define COLOR(x) (x[0])
#define HEIGHT(x) (x[1])

enum {i_color, i_height};

extern int abc_info[2];

void f(void)
{
int cur_color = COLOR(abc_info);
int this_color = abc_info[i_color];
}
```

Array types are sometimes the type of choice when sharing data between different platforms (that may use different processors) or between applications written in different languages. The relative position of each element is known, making it easy to code access mechanisms to.

Array types are characterized by their element type and by the number of elements in the array. 527

array subscript identical to

Commentary

This, along with the fact that the first element is indexed from zero, is the complete set of information needed to describe an array type.

C++

The two uses of the word *characterized* in the C++ Standard do not apply to array types. There is no other similar term applied to array types (8.3.4) in the C++ Standard.

Other Languages

Languages in the Pascal family require developers to specify the lower bound of an array type; it is not implicitly zero. Also the type used to index the array is part of the array type information; the index, in an array access, must have the same type as that given in the array declaration.

Table 527.1: Occurrence of arrays declared to have the given element type (as a percentage of all objects declared to have an array type). Based on the translated form of this book's benchmark programs.

Element Type	%	Element Type	%
`char`	17.2	`struct *`	3.7
`struct`	16.6	`unsigned int`	2.7
`float`	14.6	`enum`	2.5
other-types	10.4	`unsigned short`	2.0
`int`	8.5	`float []`	1.9
`const char`	8.0	`const char * const`	1.3
`char *`	5.1	`short`	1.1
`unsigned char`	4.4		

An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called "array of *T*". 528

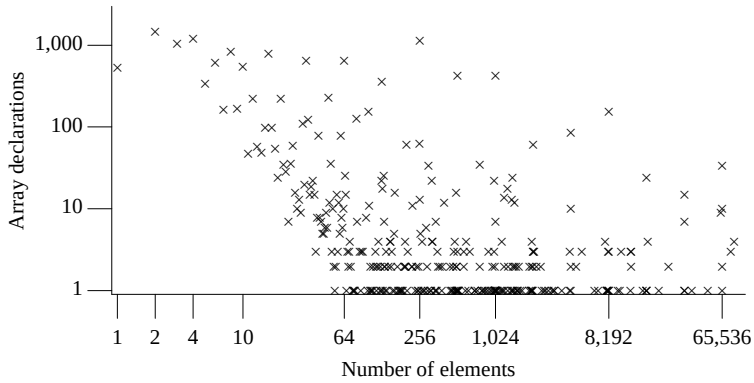

Figure 527.1: Number of arrays defined to have a given number of elements. Based on the translated form of this book's benchmark programs.

Commentary

The term *array of T* is the terminology commonly used by developers and is almost universally used in all programming languages.

C++

This usage of the term *derived from* is not applied to types in C++; only to classes. The C++ Standard does not define the term *array of T*. However, the usage implies this meaning and there is also the reference:

("array of unknown bound of T" and "array of N T") 3.9p7

---

529 The construction of an array type from an element type is called "array type derivation".

Commentary

The term *array type derivation* is used a lot in the standard to formalize the process of type creation. It is rarely heard in noncompiler writer discussions.

C++

This kind of terminology is not defined in the C++ Standard.

Other Languages

Different languages use different forms of words to describe the type creation process.

Coding Guidelines

This terminology is not commonly used outside of the C Standard, and there is rarely any need for its use.

---

530 — A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

structure type sequentially allocated objects

Commentary

This defines the term *structure type*. Structures differ from arrays in that their members

- are sequentially allocated, not contiguously allocated (there may be holes, unused storage, between them);
- may have a name;
- are not required to have the same type.

There are two ways of implementing sequential allocation; wording elsewhere reduces this to one.

member address increasing

C90

Support for a member having an incomplete array type is new in C99.

C++

C++ does not have structure types, it has class types. The keywords **`struct`** and **`class`** may both be used to define a class (and *plain old data* structure and union types). The keyword **`struct`** is supported by C++ for backwards compatibility with C.

9.2p12 *Nonstatic data members of a (non-union) class declared without an intervening `access-specifier` are allocated so that later members have higher addresses within a class object.*

C does not support static data members in a structure, or access-specifiers.

3.9.2p1 *— classes containing a sequence of objects of various types (clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities (clause 11);*

Support for a member having an incomplete array type is new in C99 and not is supported in C++.

7p3 *In such cases, and except for the declaration of an unnamed bit-field (9.6), the `decl-specifier-seq` shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration.*

The only members that can have their names omitted in C are bit-fields. Thus, taken together the above covers the requirements specified in the C90 Standard.

Other Languages

Some languages (e.g., Ada and CHILL) contain syntax which allows developers to specify the layout of members in storage (including having relative addresses that differ from their relative textual positions). Languages in the Pascal family say nothing about field ordering. Although most implementations of these languages allocate storage for fields in the order in which they were declared, some optimizers do reorder fields to optimize (either performance, or amount of generated machine code) access to them. Java says nothing about how members are laid out in a class (C structure).

Common Implementations

Some implementations provide constructs that give the developer some control over how members within a structure are laid out. The **#pack** preprocessing directive is a common extension. For example, it may simply indicate that padding between members is to be minimized, or it may take additional tokens to specify the alignments to use. While use of such extensions does not usually affect the type of a structure, developers have to take care that the same types in different translation units are associated with equivalent **#pack** preprocessing directives.

Coding Guidelines

C specifies some of the requirements on the layout of members within a structure, but not all. Possible faults arise when developers make assumptions about member layout which are not guaranteed by the standard (but happen to be followed by the particular implementation they are using). Use of layout information can also increase the effort needed to comprehend source code by increasing the amount of information that needs to be considered during the comprehension process.

represen-?? tation information using

Member layout is part of the representation of a structure type and the applicable guideline is the one recommending that no use be made of representation information. In practice developers use representation information to access members in an array-like fashion, and to treat the same area of storage as having different types.

The reason for defining a member as being part of a larger whole rather than an independent object is that it has some form of association with the other members of a structure type. This association may be

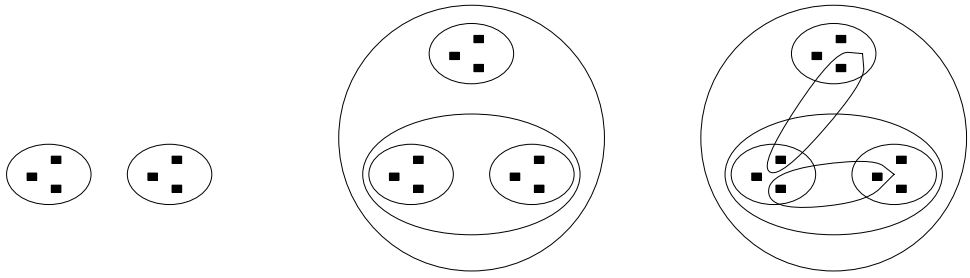

Figure 530.1: Three examples of possible member clusterings. In (a) there are two independent groupings, (b) shows a hierarchy of groupings, while in (c) it is not possible to define two C structure types that share a subset of common member (some other languages do support this functionality). The member c, for instance, might be implemented as a pointer to the value, or it may simply be duplicated in two structure types.

derived from the organization of the application domain, the internal organization of the algorithms used to implement the program, or lower-level implementation details (i.e., if several objects are frequently passed as parameters, or manipulated together the source code can be simplified by passing a single parameter or writing specific functions for manipulating these members). Deciding which structure type (category) a member belongs in is more complicated than the enumeration constant case. Structure types need not be composed of a simple list of members, they can contain instances of other structure types. It is possible to organize structure types into a hierarchy. Organizing the members of structure types is a categorization problem.

categorization
517 enumeration
set of named constants
categorization

Reorganizing existing structure type declarations to move common member subsets into a new structure definition can be costly (e.g., lots of editing of existing source to change referencing expressions). Your author's experience is that such reorganizations are rarely undertaken. The following are some of the issues that need to be considered in deciding which structure type a member belongs in:

- Is it more important to select a type based on the organization of the application's domain, on the existing internal organization of the source code, or on the expected future organization of the application domain or source code?
- Increasing the nesting of structure definitions will increase the complexity of expressions needed to access some members (an extra selection operator and member name). How much additional cognitive effort is needed to read and comprehend a longer expression containing more member names? This issue is discussed elsewhere.

member selection

- Should the number of members be limited to what is visible on a single screen or printed page? Such a limit implies that the members are somehow related, other than being in the same definition, and that developers would need to refer to different parts of the definition at the same time. If different parts of a definition do need to be viewed at the same time, then comments and blank lines need to be taken into account. It is the number of lines occupied by the definition, not the number of members, that becomes important. The issues involved in laying out definitions are discussed elsewhere. A subset of members sharing an attribute that other members do not have might be candidates for putting in another structure definition.

declaration syntax

- Does it belong to a common subset of members sharing the same attributes and types occurring within two or more structure types? Creating a common type that can be referenced, rather than duplicating members in each structure type, removes the possibility that a change to one of the common members will not be reflected in every type.

Usage

Usage information on the number of members in structure and union types and their types is given elsewhere.

limit members in struct/union
struct member type

union type overlapping members

— A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type. 531

Commentary

pointer to union members compare equal

This defines the term *union type*. The members of a union differ from a structure in that they all share the same start address; they overlap in storage. Unions are often used to interpret a storage location in different ways. There are a variety of reasons for wanting to do this, including the next two:

- Reducing the number of different objects and functions needed for accessing information in different parts of a program; for instance, it may be necessary to operate on objects having structure types that differ in only a few members. If three different types are defined, it is necessary to define three functions, each using a different parameter type:

```
struct S1 {
            int m1;
            float m2;
            long d1[3];
           };
struct S2 {
            int m1;
            float m2;
            char d2[4];
           };
struct S3 {
            int m1;
            float m2;
            long double d3;
           };

extern void f1(struct S1 *);
extern void f2(struct S2 *);
extern void f3(struct S3 *);
```

If objects operating on these three types had sufficient commonality, it could be worthwhile to define a single type and to reduce the three function definitions to a single, slightly more complicated (it has to work out which member of the union type is currently active) function:

```
struct S4 {
           int m1;
           float m2;
           union {
                 long d1[3];
                 char d2[4];
                 long double d3;
                 } m3;
           };

extern void f(struct S4 *);
```

The preceding example relies on the code knowing which union member applies in a given context. A more flexible approach is to use a member to denote the active member of the union:

```
struct node {
             enum {LEAF, UNARY, BINARY, TERNARY} type;
             struct node *left,
                         *right;
             union {
                    struct {
                           char *ident;
```

```
                        } leaf;
                struct {
                        enum {UNARYPLUS, UNARYMINUS} op;
                        struct node *operand;
                        } unary;
                struct {
                        enum {PLUS, MINUS, TIMES, DIVIDE} op;
                        struct node *left,
                                    *right;
                        } binary;
                struct {
                        struct node *test;
                        struct node *iftrue;
                        struct node *iffalse;
                        } ternary;
                } operands;
        };
```

- Another usage is the creation of a visible type punning interface:

```
union {
      int m1; /* Assume int is 24 bits. */
      struct {
              char c1; /* Assumes char is 8 bits and similarly aligned. */
              char c2;
              char c3;
             } m2;
      } x;
```

  Here the three bytes of an object, having type **int**, may be manipulated by referencing `c1`, `c2`, and `c3`. The same effect could have been achieved by using pointer arithmetic. In both cases the developer is making use of implementation details, which may vary considerably between implementations.

C++

Each data member is allocated as if it were the sole member of a struct. 9.5p1

This implies that the members overlap in storage.

— unions, which are classes capable of containing objects of different types at different times, 9.5; 3.9.2p1

In such cases, and except for the declaration of an unnamed bit-field (9.6), the `decl-specifier-seq` shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration. 7p3

The only members that can have their names omitted in C are bit-fields. Thus, taken together the preceding covers the requirements specified in the C Standard.

Other Languages

Pascal does not have an explicit **union** type. However, it does support something known as a *variant record*, which can only occur within a **record** (**struct**) definition. These variant records provide similar, but more limited, functionality (only one variant record can be defined in any record, and it must occur as the last field).

Java's type safety system would be broken if it supported the functionality provided by the C union type and it does not support this form of construct (although the Java library does contain some functions

for converting values of some types, to and from their bit representation— e.g., `floatToIntBits` and `intBitsToFloat`).

The Fortran **EQUIVALENCE** statement specifies that a pair of identifiers share storage. It is possible to equivalence a variable having a scalar type with a particular array element of another object, and even for one array's storage to start somewhere within the storage of a differently named array. This is more flexible than C, which requires that all objects start at the same storage location.

Algol 68 required implementations to track the last member assigned to during program execution. It was possible for the program to do a runtime test, using something called a conformity clause, to find out the member last assigned to.

Coding Guidelines

Union types are created for several reasons, including the following:

- Reduce the number of closely related functions by having one defined with a parameter whose union type includes all of the types used by the corresponding, related functions.
- Access the same storage locations using different types (often to access individual bytes).
- Reduce storage usage by overlaying objects having different types whose access usage is mutually exclusive in the shared storage locations.

Merging closely related functions into a single function reduces the possibility that their functionality will diverge. The extent to which this benefit exceeds the cost of the increase in complexity (and hence maintenance costs) of the single function can only be judged by the developer.

type punning
union

union member when written to

Accessing the same storage locations using different types depends on undefined and implementation-defined behaviors. The standard only defines the behavior if the member being read from is the same member that was last written to. Performing such operations is often unconditionally recommended against in coding guideline documents. However, use of a union type is just one of the ways of carrying out type punning. In this case the recommendation is not about the use of union types; it is either about making use of undefined and implementation-defined behaviors, or the use of type punning. (The guideline recommendation on the use of representation information is applicable here.)

representation information using ??

If developers do, for whatever reason, want to make use of type punning, is the use of a union type better than the alternatives (usually casting pointers)? When a union type is used, it is usually easy to see which different types are involved by looking at the definitions. When pointers are used, it is usually much harder to obtain a complete list of the types involved (all values, and the casts involved, assigned to the pointer object need to be traced). Union types would appear to make the analysis of the code much easier than if pointer types are used.

declaration interpretation of identifier

Using union types to reduce storage usage by overlaying objects having different types is a variation of using the same object to represent different semantic values. The same guideline recommendations are applicable in both cases.

A union type containing two members that have compatible type might be regarded as suspicious. However, the types used in the definition of the members may be typedef names (one of whose purposes is to hide details of the underlying type). Instances of a union type containing two or more members having a compatible type, where neither is a typedef name, are not sufficiently common to warrant a guideline.

Example

```
1  union U_1 {
2              int m_1;
3              int m_2;
4             };
5
6  typedef int APPLES;
```

```
7   typedef int ORANGES;
8
9   union U_2 {
10              APPLES a_count;
11              ORANGES o_count;
12             };
13
14  union U_3 {
15              float f1;
16              unsigned char f_bytes[sizeof(float)];
17             };
```

532 — A *function type* describes a function with specified return type. function type

Commentary

This defines the term *function type*. A function type is created either by a function definition, the definition of an object or typedef name having type pointer-to function type, or a function declarator. function definition syntax function declarator return type

According to this definition, the parameters are not part of a function's type. The type checking of the arguments in a function call, against the corresponding declaration, are listed as specific requirements under the function call operator rather than within the discussion on types. However, the following sentence includes parameters as part of the characterization of a function type. function call

C++

> — *functions, which have parameters of given types and return* ***void*** *or references or objects of a given type, 8.3.5;* 3.9.2p1

The parameters, in C++, need to be part of a function's type because they may be needed for overload resolution. This difference is not significant to developers using C because it does not support overloaded functions.

Other Languages

Functions, as types, are not common in other algorithmic languages. Although nearly every language supports a construct similar to C functions, and may even call them function (or procedure) types, it is not always possible to declare objects to refer to these types (like C supports pointers to functions). Some languages do allow references to function types to be passed as arguments in function calls (this usage invariable requires some form of prototype definition). Then the called function is able to call the function referred to by the corresponding parameter. In functional languages, function types are an integral part of the design of the language type system. A Java method could be viewed as a function type.

533 A function type is characterized by its return type and the number and types of its parameters.

Commentary

The **inline** function specifier is not part of the type. function specifier syntax

C++

C++ defines and uses the concept of function *signature* (1.3.10), which represents information amount the number and type of a function's parameters (not its return type). The two occurrences of the word *characterizes* in the C++ Standard are not related to functions.

Usage

Usage information on function return types is given elsewhere (see Table **??**) as is information on parameters (see Table **??**).

function returning T

A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called "function returning *T*". 534

Commentary

The term *function returning T* is a commonly used term. The term *function taking parameters* is heard, but not as often.

C++

The term *function returning T* appears in the C++ Standard in several places; however, it is never formally defined.

Other Languages

Different languages use a variety of terms to describe this kind of construct.

The construction of a function type from a return type is called "function type derivation". 535

Commentary

This terminology may appear in the standard, but it is very rarely heard in discussions.

C++

There is no such definition in the C++ Standard.

---

footnote 35

35) **CHAR_MIN**, defined in **<limits.h>**, will have one of the values 0 or **SCHAR_MIN**, and this can be used to distinguish the two options. 536

Commentary

CHAR_MAX char if treated as signed integer

Similarly, CHAR_MAX will have one of two values that could be used to distinguish the two options.

C++

The C++ Standard includes the C90 library by reference. By implication, the preceding is also true in C++.

Example

```
1  #include <limits.h>
2
3  #if CHAR_MIN == 0
4  /* ... */
5  #elif CHAR_MIN == SCHAR_MIN
6  /* ... */
7  #else
8  #error Broken implementation
9  #endif
```

---

char separate type

Irrespective of the choice made, **char** is a separate type from the other two and is not compatible with either. 537

Commentary

types different 508 even if same representation

This sentence calls out a special case of the general rule that any two basic types are different. In most cases, **char** not being compatible with its *matching* type is not noticeable (an implicit conversion is performed). However, while objects having different character types may be assigned to each other, a pointer-to **char** and a pointer to any other character type may not.

C++

Plain **char**, **signed char**, *and* **unsigned char** *are three distinct types.*

Common Implementations

Some implementations have been known to not always honor this requirement, treating **char** and its *matching* character type as if they were the same type.

Coding Guidelines

A common developer expectation is that the type **char** will be treated as either of the types **signed char**, or an **unsigned char**. It is not always appreciated that the types are different. Apart from the occasional surprise, this incorrect assumption does not appear to have any undesirable consequences.

Example

```
1  char p_c,
2       *p_p_c = &p_c;
3  signed char s_c,
4              *p_s_c = &s_c;
5  unsigned char u_c,
6                *p_u_c = &u_c;
7
8  void f(void)
9  {
10 p_c = s_c;
11 p_c = u_c;
12
13 p_p_c = p_s_c;
14 p_p_c = p_u_c;
15 }
```

---

538 36) Since object types do not include incomplete types, an array of incomplete type cannot be constructed. footnote 36

Commentary

Object types do not include function types either. But they do include pointer-to function types.

C++

Incompletely-defined object types and the void types are incomplete types (3.9.1). 3.9p6

The C++ Standard makes a distinction between incompletely-defined object types and the **void** type. 475 object types

The declared type of an array object might be an array of incomplete class type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type. 3.9p7

The following deals with the case where the size of an array may be omitted in a declaration:

When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays can be omitted only for the first member of the sequence. 8.3.4p3

Arrays of incomplete structure and union types are permitted in C++.

```
{
struct st;
typedef struct st_0 A[4]; /* Undefined behavior */
                          // May be well- or ill-formed
typedef struct st_1 B[4]; /* Undefined behavior */
                          // May be well- or ill-formed
struct st_0 {             /* nothing has changed */
              int mem;    // declaration of A becomes well-formed
             };
}                         /* nothing has changed */
                          // declaration of B is now known to be ill-formed
```

Other Languages

Most languages require that the size of the element type of the array be known at the point the array type is declared.

---

pointer type
referenced type

— A *pointer type* may be derived from a function type, an object type, or an incomplete type, called the *referenced type*. 539

Commentary

This defines the terms *pointer type* (a term commonly used by developers) and *referenced type* (a term not commonly used by C developers). Pointers in C have maximal flexibility. They can point at any kind of type (developers commonly use the term *point at*).

C++

C++ includes support for what it calls *reference* types (8.3.2), so it is unlikely to use the term *referenced type* in this context (it occurs twice in the standard). There are requirements in the C++ Standard (5.3.1p1) that apply to pointers to object and function types, but there is no explicit discussion of how they might be created.

Other Languages

Some languages do not include pointer types. They were added to Fortran in its 1991 revision. They are not available in Cobol. Some languages do not allow pointers to refer to function types, even when the language supports some form of function types (e.g., Pascal). Java does not have pointers, it has references.

Coding Guidelines

Use of objects having pointer types is often considered to be the root cause of many faults in programs written in C. Some coding guideline documents used in safety-critical application development prohibit the use of pointers completely, or severely restrict the operations that may be performed on them. Such prohibitions are not only very difficult to enforce in practice, but there is no evidence to suggest that the use of alternative constructs reduces the number of faults.

string literal
static storage duration

String literals have type pointer to **char**; an array passed as an argument will be converted to a pointer to its element type, and without the ability to declare parameters having a pointer type, significantly more information has to be passed via file scope objects (pointers are needed to implement the parameter-passing concept of call-by address).

Table 539.1: Occurrence of objects declared using a given pointer type (as a percentage of all objects declared to have a pointer type). Based on the translated form of this book's benchmark programs.

Pointed-to Type	%	Pointed-to Type	%
`struct`	66.5	`struct *`	1.8
`char`	8.0	`int`	1.8
`union`	6.0	`const char`	1.3
other-types	5.5	`char *`	1.2
`void`	3.3	str | str	
`unsigned char`	2.6	`_double | _double`	
`unsigned int`	2.2	`_double | _double`	

540 A pointer type describes an object whose value provides a reference to an entity of the referenced type.

pointer type describes a

Commentary

The value in a pointer object is a reference. Possible implementations of a reference include an address in storage, or an index into an array that gives the actual address. In C the value of a pointer object is rarely called a *reference*. The most commonly used terminology is *address*, which is what a pointer value is on most implementations. Sometimes the term *pointed-to object* is used.

Other Languages

Other languages use a variety of terms to refer to the value stored in an object having pointer type. Java uses the term *reference* exclusively; it does not use the term *pointer* at all. The implementation details behind a Java reference are completely hidden from the developer.

Common Implementations

Nearly every implementation known to your author represents a reference using the address of the referenced object only (in some cases the address may not be represented as using a single value). Some implementations use a, so-called, *fat pointer* representation.[3, 26] These implementations are usually intended for use during program development, where information on out-of-bounds storage accesses is more important than speed of execution. A fat pointer includes information on the pointed-to object such as its base address and the number of bytes in the object.

pointer segmented architecture

In the Model Implementation C Checker[18] a function address is implemented as two numbers. One is an index into a table specifying the file containing the translated function definition and the other is an index into a table specifying the offset of the executable code within that file. The IBM AIX compiler[14] also uses a function descriptor, not the address of the generated code.

The Java virtual machine does not include the concept of pointer (or address). It includes the concept of a reference; nothing is said about how such an entity is implemented. A C translator targeted at this host would have to store a JVM reference in an object having pointer type.

Some processors differentiate between different kinds of main storage. Access to different kinds of storage are faster/slower, or accesses to particular storage areas may only be made via particular registers. Translators for such processors usually provide keywords, enabling developers to specify which kinds of storage pointers will be pointing at.

Some implementations use different pointer representations (they usually vary in the number of bytes used), depending on how the pointer is declared. For instance, the keywords **`near`**, **`far`**, and **`huge`** are sometimes provided to allow developers to specify the kind of representation to use.

The HP C/iX translator[23] supports a short pointer (32-bit) and long pointer (64-bit). A long pointer has two halves, 32 bits denoting a process-id and 32 bits denoting the offset within that process address space. The Unisys A Series implementation[31] represents pointers as integer values. A pointer to **`char`** is the number of bytes from the start of addressable storage (for that process), not the logical address of the storage location. A pointer to the types **`int`** and **`float`** is the number of words (6 bytes) from the start of storage (the unit of measurement for other types is the storage alignment used for the type).

pointer compressing members

Zhang and Gupta[32] developed what they called the *common prefix* transformation, which compresses a 32-bit pointer into 15 bits (this is discussed elsewhere). There has been some research[27] investigating the use of Gray codes, rather than binary, to represent addresses. Successive values in a Gray code differ from each other in a single bit. This property can have advantages in high-performance, or low-power (electrical) situations when sequencing through a series of storage locations.

A pointer type derived from the referenced type *T* is sometimes called "pointer to *T*". 541

Commentary

The term *pointer to T* is commonly used by developers, and is almost universally used for all programming languages.

Other Languages

This term is almost universally used for all languages that contain pointer types.

The construction of a pointer type from a referenced type is called "pointer type derivation". 542

Commentary

The term *pointer type derivation* is used a lot in the standard to formalize the process of type creation. It is rarely heard outside of the committee and compiler writer discussions.

C++

The C++ Standard does not define this term, although the term *derived-declarator-type-list* is defined (8.3.1p1).

Other Languages

Different languages use different forms of words to describe the type creation process.

These methods of constructing derived types can be applied recursively. 543

Commentary

There are two methods of constructing derived types in the visible source; either a typedef name can be used, or the declaration of the type can contain nested declarations. For instance, an array of array of *type T* might be declared as follows:

```
1  typedef int at[10];
2  at obj_x[5];
3
4  int obj_y[10][5];
```

limit type complexity

The standard specifies minimum limits on the number of declarators that an implementation is required to support.

Other Languages

footnote 121

Most languages support more than one level of type derivation. Many languages support an alternative method of declaring multidimensional arrays, where **[ ]** are not treated as an operator. Structure types were added in Fortran 90, and only support a single nesting level.

---

scalar types

Arithmetic types and pointer types are collectively called *scalar types*. 544

Commentary

This defines the term *scalar type*. It is commonly used by developers and it is also used in many other programming languages. The majority of operations in C act on objects and values having a scalar type.

C++

Arithmetic types (3.9.1), enumeration types, pointer types, and pointer to member types (3.9.2), and `cv-qualified` *versions of these types (3.9.3) are collectively called scalar types.* 3.9p10

While C++ includes type qualifier in the definition of scalar types, this difference in terminology has no impact on the interpretation of constructs common to both languages.

Other Languages

The term *scalar type* is used in many other languages. Another term that is sometimes heard is *simple type*.

Common Implementations

Many processors only contain instructions that can operate on values having a scalar type. Operations on aggregate types are broken down into operations on their constituent scalar components. Many implementations only perform some optimizations at the level of scalar types (components of derived types having a scalar type are considered for optimization, but the larger whole is not). For instance, the level of granularity used to allocate values to registers is often at the level of scalar types.

Coding Guidelines

Pointer types commonly occur in many of the same contexts as arithmetic types. Having guideline recommendations that apply to both is often a useful generalization and reduces the number of special cases. The following is a meta-guideline recommendation.

Rev 544.1 Where possible coding guidelines shall try to address scalar types, rather than just arithmetic types.

---

545 Array and structure types are collectively called *aggregate types*.[37)] aggregate type

Commentary

This defines the term *aggregate type*. This terminology is often incorrectly used by developers. An aggregate type includes array types but does not include union types.

C++

An aggregate is an array or a class (clause 9) with no user-declared constructors (12.1), no private or protected non-static data members (clause 11), no base classes (clause 10), and no virtual functions (10.3). 8.5.1p1

Class types in C++ include union types. The C definition of aggregate does not include union types. The difference is not important because everywhere that the C++ Standard uses the term *aggregate* the C Standard specifies aggregate and union types.

The list of exclusions covers constructs that are in C++, but not C. (It does not include static data members, but they do not occur in C and are ignored during initialization in C++.) There is one place in the C++ Standard (3.10p15) where the wording suggests that the C definition of aggregate is intended.

Coding Guidelines

Although the logic behind the term *aggregate type* is straight-forward, a type made up of more than one object (the array type having one element, or the structure type having one member, is considered to be a degenerate case), is a categorization of types that is rarely thought about by developers. In most developer discussions, array and structure types are not thought of as belonging to a common type category. 553 type category

The term *aggregate type* is commonly misused. Many developers assume that it includes the union types in its definition; they are not aware that array types are included in the definition. To avoid confusion, this term is probably best avoided in coding guideline documents.

array
unknown size

An array type of unknown size is an incomplete type. 546

Commentary

array
incomplete type
object
type complete by end

The size is unknown in the sense that the number of elements is not known. The term *incomplete array type* is often used. Objects with no linkage cannot have an incomplete type.

Other Languages

Some languages include the concept of an array having an unknown number of elements. This usually applies only to the type of a parameter in which arrays with different numbers of elements are passed. There is often a, language-provided, mechanism for finding the number of elements in the array actually passed as an argument. In other cases it is the developer's responsibility to pass that information as an argument, along with the array itself. In Java all declarations of objects having an array type omit the number of elements. The actual storage is allocated during program execution using the operator **new**.

array
type completed by

It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). 547

Commentary

A typedef name that has an incomplete array type cannot be completed. However, an object definition, whose type specifier is the typedef name, can complete this type for its definition. An initializer appearing as part of the object's definition provides a mechanism for a translator to deduce the size of the array type. The size, actually the number of elements, may also be implicitly specified, if there is no subsequent declaration that completes the type, when the end of the translation unit is reached.

array of unknown size
initialized
object definition
implicit
array type
adjust to pointer to

In the case of parameters their type is converted to a pointer to the element type.

C++

3.9p7 *The declared type of an array object might be an array of unknown size and therefore be incomplete at one point in a translation unit and complete later on;*

Which does not tell us how it got completed. Later on in the paragraph we are given the example:

3.9p7
```
extern int arr[];   // the type of arr is incomplete

int arr[10];        // now the type of arr is complete
```

which suggests that an array can be completed, in a later declaration, by specifying that it has 10 elements. :-)

Other Languages

Fortran supports the declaration of subroutine parameters taking array types, whose size is not known at translation time. Array arguments are passed by reference, so the translator does not need to know their size. Developers usually pass the number of elements as another parameter to the subroutine.

known constant
size

A type has *knownconstantsize* if the type is not incomplete and is not a variable length array type. 548

Commentary

The sentence was added by the response to DR #312 and clarifies that *known constant size* is to be interpreted as a technical term involving types and not the kind of expressions that an implementation may chose to treat as being constants.

constant
syntax

structure
incomplete type
union
incomplete type

A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. 549

Commentary

Incomplete structure and union types are needed to support self-recursive and mutually recursive declarations involving more than one such structure or union. These are discussed in subclause 6.7.2.3.

type
contents defined once

Other Languages

A mechanism for supporting mutual recursion in type definitions is invariably provided in languages that support some form of pointer type. A variety of special rules is used by different languages to allow mutually referring data types to be defined.

Example

```
1  struct U {
2             int M1;
3             struct U *next;
4            };
5
6  struct S;  /* Members defined later. */
7
8  struct T {
9             int m1;
10            struct S *s_m;
11           };
12 struct S { /* S now completed. */
13            long m1;
14            struct T *t_m;
15           };
```

The two mutually referential structure types could not be declared without the original, incomplete declaration of S.

---

550 It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

incomplete type completed by

Commentary

A definition of the same tag name in a different scope is a different definition and does not complete the declaration in the outer scope.

C++

A class type (such as "`class X`") might be incomplete at one point in a translation unit and complete later on; 3.9p7

An example later in the same paragraph says:

```
class X;                // X is an incomplete type

struct X { int i; };    // now X is a complete type
```

3.9p7

The following specifies when a class type is completed; however, it does not list any scope requirements.

A class is considered a completely-defined object type (3.9) (or complete type) at the closing `}` of the `class-specifier`. 9.2p2

In practice the likelihood of C++ differing from C, in scope requirements on the completion of types, is small and no difference is listed here.

statements integrating information between

Coding Guidelines

Having important references to an identifier close together in the visible source code has a number of benefits. In the case of mutually recursive structure and union types, this implies having the declarations and definitions adjacent to each other.

Rev 550.1

The completing definition, of an incomplete structure or union type, shall occur as close to the incomplete declaration as permitted by the rules of syntax and semantics.

Example

```
1  struct INCOMP_TAG;              /* First type declaration. */
2  struct INCOMP_TAG *gp;          /* References first type declaration. */
3  extern void f(struct INCOMP_TAG /* Different scope. */);
4
5  void g(void)
6  {
7  struct INCOMP_TAG {int mem1;} *lp; /* Different type declaration. */
8
9  lp = gp; /* Not compatible pointers. */
10 }
11
12 void h(void)
13 {
14 struct INCOMP_TAG; /* Different type declaration. */
15 struct INCOMP_TAG *lp;
16
17 lp = gp; /* Not compatible pointers. */
18 }
19
20 struct INCOMP_TAG {
21                     int m_1;
22                    };
```

---

derived declarator types

Array, function, and pointer types are collectively called *derived declarator types*. 551

Commentary

This defines the term *derived declarator types*. This term is used a lot in the standard to formalize the process of type creation. It is rarely heard outside of committee and translator writer discussions.

C++

There is no equivalent term defined in the C++ Standard.

Other Languages

Different languages use different terms to describe the type creation process.

Coding Guidelines

This terminology is not commonly used outside of the C Standard and its unfamiliarity, to developers, means there is little to be gained by using it in coding guideline documents.

A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*. 552

Commentary

This defines the term *declarator type derivation*. This term does not appear to be used anywhere in the standard, except the index and an incorrect forward reference.

C++

There is no equivalent definition in the C++ Standard, although the description of compound types (3.9.2) provides a superset of this definition.

---

553 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types. type category

Commentary

This defines the term *type category*. Other terms sometimes also used by developers, which are not defined in the standard, are *outermost type* and *top level type*. An object is commonly described as an *array-type*, a *pointer-type*, a *structure-type*, and so on. Without reference to its constituents. But the term *type category* is rarely heard in developer discussions.

The following was included in the response to DR #272:

DR #272

Committee Discussion *(for history only)*

The committee wishes to keep the term "type category" for now, removing the term "type category" from the next revision of the standard should be considered at that time.

C++

The term *outermost level* occurs in a few places in the C++ Standard, but the term *type category* is not defined.

Other Languages

The concept denoted by the term *type category* exists in other languages and a variety of terms are used to denote it.

Coding Guidelines

The term *type category* is not commonly used by developers (it only occurs in five other places in the standard). Given that terms such as *outermost type* are not commonly used either, it would appear that there is rarely any need to refer to the concept denoted by these terms. Given that there is no alternative existing common practice there is no reason not to use the technically correct term; should a guidelines document need to refer to this concept.

---

554 Any type so far mentioned is an *unqualified type*. unqualified type

Commentary

This defines the term *unqualified type*. An unqualified type is commonly referred to, by developers, as just the *type*, omitting the word unqualified. The suffix qualified is only used in discussions involving type qualifiers, to avoid ambiguity.

C++

In C++ it is possible for the term *type* to mean a qualified or an unqualified type (3.9.3).

Coding Guidelines

It is common practice to use the term *type* to mean the unqualified type. Unqualified types are much more commonly used than qualified types. While the usage of the term *type* might be generally accepted by C developers to mean an unqualified type, this usage is not true in C++. Guidelines that are intended to be applied to both C and C++ code will need to be more precise in their use of terminology than if they were aimed at C code only.

Table 554.1: Occurrence of qualified types as a percentage of all (i.e., qualified and unqualified) occurrences of that kind of type (e.g., * denotes any pointer type, **struct** any structure type, and *array of* an array of some type). Based on the translated form of this book's benchmark programs.

Type Combination	%	Type Combination	%
array of **const**	26.7	**const** *	0.4
const integer-type	4.8	**const union**	0.3
const real-type	2.7	**volatile struct**	0.1
* **const**	2.6	**volatile** integer-type	0.1
const struct	2.4	* **volatile**	0.1

qualified type versions of

Each unqualified type has several *qualified versions* of its type,[38)] corresponding to the combinations of one, two, or all three of the **const**, **volatile**, and **restrict** qualifiers. 555

Commentary

This defines the term *qualified version* of a type. In the case of structure, and union types, qualifiers also qualify their member types. Type qualifiers provide a means for the developer to provide additional information about the properties of an object. In general these properties relate to issues involved with the optimization of C programs.

C90

The **noalias** qualifier was introduced in later drafts of what was to become C90. However, it was controversial and there was insufficient time available to the Committee to resolve the issues involved. The noalias qualifier was removed from the document, prior to final publication. The **restrict** qualifier has the same objectives as **noalias**, but specifies the details in a different way.

Support for the **restrict** qualifier is new in C99.

C++

3.9.3p1 *Each type which is a cv-unqualified complete or incomplete object type or is* **void** *(3.9) has three corresponding cv-qualified versions of its type: a* `const-qualified` *version, a* `volatile-qualified` *version, and a* `const-volatile-qualified` *version.*

The **restrict** qualifier was added to C99 while the C++ Standard was being finalized. Support for this keyword is not available in C++.

Other Languages

Pascal uses the **packed** keyword to indicate that the storage occupied by a given type should be minimized (packed, so there are no unused holes). Java has 10 different modifiers; not all of them apply directly to types. Some languages contain a keyword that enables an object to be defined as being read-only.

Common Implementations

The standard provides a set of requirements that an implementation must honor for an object with a given qualified type. The extent to which a particular translator makes additional use of the information provided varies.

Table 555.1: Occurrence of type qualifiers on the outermost type of declarations occurring in various contexts (as a percentage of all type qualifiers on the outermost type in these declarations). Based on the translated form of this book's benchmark programs.

Type Qualifier	Local	Parameter	File Scope	**typedef**	Member	Total
const	18.5	4.3	50.8	0.0	1.2	74.8
volatile	1.6	0.1	3.0	0.1	20.4	25.2
volatile const	0.0	0.0	0.0	0.0	0.0	0.0
Total	20.1	4.4	53.8	0.1	21.6	

556 The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[39)]

qualifiers representation and alignment

Commentary

Qualifiers apply to objects whose declarations include them. They do not play any part in the interpretation of a value provided by a type, but they participate in the type compatibility rules.

Other Languages

This statement can usually be applied to qualifiers defined in other languages.

Common Implementations

Objects declared using a qualified type may have the same representation and alignment requirements, but there are no requirements specifying where they might be allocated in storage. Some implementations chose to allocate differently qualified objects in different areas of storage. For instance, const-qualified objects may be placed in read-only storage; volatile-qualified objects may be mapped to special areas of storage associated with I/O ports.

557 A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

derived type qualification

Commentary

For instance, a type denoting a const-qualified **char** does not also result in a pointer to it to also being const-qualified, although the pointed-to type retains its **const** qualifier.

A structure type containing a member having a qualifier type does not result in that type also being so qualified. However, an object declared to have such a structure type will share many of the properties associated with objects having the member's qualified type when treated as a whole. For instance, the presence of a member having a const-qualified type, in a structure type, prevents an object declared using it from appearing as the left operand of an assignment operator. However, the fact that one member has a const-qualified type does not affect the qualification of any other members of the same structure type.

Other Languages

This specification usually applies to other languages that support some form of type qualifiers, or modifiers.

Coding Guidelines

Inexperienced developers sometimes have problems distinguishing between constant pointers to types and pointers to constant types. Even the more experienced developer might be a little confused over the following being conforming:

```
1  void f(int * const a[])
2  {
3  a++; /* Type of a is pointer to constant pointer to int. */
4  }
```

Rev 557.1 Array and pointer types that include a qualifier shall be checked to ensure that the type that is so qualified is the one intended by the original author.

Translators will probably issue a diagnostic for those cases in which a **const** qualifier was added where it was not intended (e.g., because of an attempt to modify a value). However, translators are not required to issue a diagnostic for a **const** qualifier that has been omitted (unless there is a type compatibility associated with the assignment, or argument passing). Some static analysis tools[7,8] diagnose declarations where a **const** qualifier could be added to a type without violating any constraints.

Example

In the following declarations x1 is not a const-qualified structure type. However, one of its members is const-qualified. The member x1.m2 can be modified. y1 is a const-qualified structure type. The member y1.m2 cannot be modified.

```
typedef const int CI;

CI *p;   /* The pointed-to type is qualified, not the pointer. */
CI a[3]; /*
          * a is made up of const ints, it is not
          * possible to qualify the array type.
          */

struct S1 {
           const int m1;
           long m2;
          } x1, x2;

const struct S2 {
                 const int m1;
                 long m2;
                } y1;

void f(void)
{
x1 = x2; /* Constraint violation. */
}
```

What are the types in:

```
typedef int *I;

I const p1; /* A const qualified pointer to int. */
const I p2; /* A const qualified pointer to int. */
```

---

pointer to void same representation and alignment as

A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type.[39)] 558

Commentary

This is a requirement on the implementation. In its role as a generic container for any pointer value, a pointer to **void** needs to be capable of holding the *hardest* reference to represent. Experience has shown that, in those cases where different representations are used for pointers to different types, this is usually the pointer to character type.

Prior to the publication of C90, pointers to character types were often used to perform the role that pointer to **void** was designed to fill. That is, they were the pointer type used to represent the concept of pointer to any type, a generic pointer type (through suitable casting, which is not required for pointer to **void**). Existing code that uses pointer to character type as the generic pointer type can coexist with newly written code that uses pointer to **void** for this purpose.

generic pointer 523

Other Languages

Most languages that contain pointer types do not specify a pointer type capable of representing any other pointer type. Although pointer to character type is sometimes used by developers for this purpose.

Coding Guidelines

This C requirement is intended to allow existing code to coexist with newly written code using pointer to **void**. Mixing the two pointer types in newly written code serves no useful purpose. The fact that the two

kinds of pointers have the same representation requirements does not imply that they represent a reference to the same object with the same pattern of bits (any more than two pointers of the same type are required to). The guideline recommendation dealing with the use of representation information is applicable here.

?? representation information using

559 Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements.

pointer to qualified/unqualified types

Commentary

This is a requirement on the implementation.

The representation and alignment of a type is specified as being independent of any qualifiers that might appear on the type. Since the pointed-to type has these properties, it might be expected that pointers to them would also have these properties.

556 qualifiers representation and alignment

Common Implementations

This requirement on the implementation rules out execution-time checking of pointer usage by using different representations for pointers to qualified and unqualified types.

The C model of storage is of a flat (in the sense of not having any structure to it) expanse into which objects can be allocated. Some processors have disjoint storage areas (or *banks*). They are disjoint in that either different pointer representations are required to access the different areas, or because execution of a special instruction causes subsequent accesses to reference a different storage area. The kind of storage referred to by a pointer value, may be part of the encoding of that value, or the processor may have state information that indicates which kind of storage is currently the default to be accessed, or the kind of storage to access may be encoded in the instruction that performs the access.

banked storage

The IAR PICmicro compiler[11] provides access to more than 10 different kinds of banked storage. Pointers to this storage can be 1, 2, or 3 bytes in size.

Coding Guidelines

The fact that the two kinds of pointers have the same representation requirements does not imply that they represent a reference to the same object with the same pattern of bits (any more than two pointers of the same type are required to). The guideline recommendation dealing with the use of representation information is applicable here.

?? representation information using

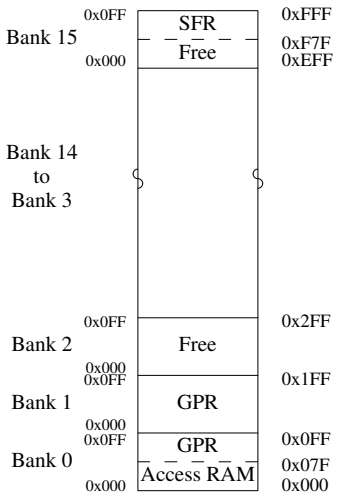

Figure 559.1: Data storage organization for the PIC18CXX2 devices[21] The 4,096 bytes of storage can be treated as a linear array or as 16 banks of 256 bytes (different instructions and performance penalties are involved). Some storage locations hold Special Function Registers (SFR) or General Purpose Registers (GPR). *Free* denotes storage that does not have a preassigned usage and is available for general program use.

alignment pointer to structures representation pointer to structures

All pointers to structure types shall have the same representation and alignment requirements as each other. 560

Commentary

This is a requirement on the implementation. It refers to the pointer type, not the pointed-to type. This specification is redundant in that it can be deduced from other requirements in the standard. A translation unit can define a pointer to an incomplete type, where no information on the pointed-to type is provided within that translation unit. In:

```
1  #include <stdlib.h>
2
3  extern struct tag *glob_p;
4
5  int f(void)
6  {
7  if (glob_p == NULL)
8     return 1;
9
10 glob_p = (struct tag *)malloc(8);
11 return 0;
12 }
```

a translator knows nothing about the pointed-to type (apart from its tag name, and it would be an unusual implementation that based alignment decisions purely on this information). If pointers to different structure types had different representations and alignments, the implementation would have to delay generating machine code for the function `f` until link-time.

C90

This requirement was not explicitly specified in the C90 Standard.

C++

The C++ Standard follows the C90 Standard in not explicitly stating any such requirement.

Other Languages

Most languages do not get involved in specifying details of pointer representation and alignment.

Coding Guidelines

representation information using ??

The fact that the two kinds of pointers have the same representation requirements does not mean that they represent a reference to the same object with the same pattern of bits (any more than two pointers of the same type are required to). The guideline recommendation dealing with the use of representation information is applicable here.

alignment pointer to unions representation pointer to unions

All pointers to union types shall have the same representation and alignment requirements as each other. 561

Commentary

alignment 560 pointer to structures

The chain of deductions made for pointers to structure types also apply to pointer-to union types.

C90

This requirement was not explicitly specified in the C90 Standard.

C++

The C++ Standard follows the C90 Standard in not explicitly stating any such requirement.

Other Languages

Most languages do not get involved in specifying details about pointer representation and alignment.

Coding Guidelines

The fact that the two kinds of pointers have the same representation requirements does not mean that they represent a reference to the same object with the same pattern of bits (any more than two pointers of the same type are required to). The guideline recommendation dealing with the use of representation information is applicable here.

?? representation information using

562 Pointers to other types need not have the same representation or alignment requirements.

alignment pointers

Commentary

Although many host processors use the same representation for all pointer types, this is not universally true, and this permission reflects this fact.

C++

The value representation of pointer types is implementation-defined. 3.9.2p3

Other Languages

Most languages do not get involved in specifying details of pointer representation and alignment.

Common Implementations

Some processors use what is sometimes known as word *addressing*. This hardware characteristic may, or may not, result in some pointer types having different representations.

word addressing

563 37) Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

footnote 37

Commentary

As a phrase, the term *aggregate type* is open to several interpretations. Experience shows that developers sometimes classify union types as being aggregate types. This thinking is based on the observation that structure and union types often contain many different types— an aggregate of types. However, the definition used by the Committee is based on there being an aggregate of objects. Although an object having a union type can have many members, only one of them represents a value at any time (an object having a structure or array type is usually capable of representing several values at the same time).

The phrase *one member at a time* is a reference to the fact that the value of at most one member can be stored in an object having a union type at any time.

union member at most one stored

C++

The C++ Standard does include union types within the definition of aggregate types, 8.5.1p1. So, this rationale was not thought applicable by the C++ Committee.

Other Languages

Union types, as a type category, are unique to C (and C++), so this issue does not occur in other languages.

553 type category

564 38) See 6.7.3 regarding qualified array and function types.

footnote 38

565 39) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

footnote 39

Commentary

The text of this footnote is identical to footnote 31; however, the rationale behind it is different. Type qualifiers did not exist prior to C90. Supporting a degree of interchangeability allows developers to gradually introduce type qualifiers into their existing source code without having to modify everything at once. Also source code containing old-style function declarations continues to exist. There is the possibility of pointers to qualified types being passed as arguments to such functions.

509 footnote 31

Other Languages

Most languages that contain type qualifiers, or modifiers, do not get involved in this level of implementation detail.

Coding Guidelines

Qualified and qualified version of the same type may appear as members of unions. This interchangeability of members almost seems to invite the side-stepping of the qualifier.

EXAMPLE 1 The type designated as "`float *`" has type "pointer to `float`". Its type category is pointer, not a floating type. The const-qualified version of this type is designated as "`float * const`" whereas the type designated as "`const float *`" is not a qualified type— its type is "pointer to const-qualified `float`" and is a pointer to a qualified type. 566

EXAMPLE 2 The type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type `float`. Its type category is array. 567

Forward references: compatible type and composite type (6.2.7), declarations (6.7). 568

References

1. Altrium BV. *C166/ST10 v8.0 C Cross-Compiler User's Guide*. Altrium BV, 2003.
2. Anon. The international obfuscated C code contest. www.ioccc.org, 2003.
3. T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.
4. G. Boole. *An Investigation of the Laws of Thought*. Dover Publications, 1973.
5. D. J. Boorstin. *The Discoverers*. Phoenix Press, 1983.
6. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. *ACM SIGPLAN Notices*, 34(7):96–103, July 1999.
7. J. S. Foster and et al. CQUAL *User's Guide*. University of California, Berkeley, version 0.9 edition, Jan. 2002.
8. J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):1035–1087, Nov. 2006.
9. J. M. Gravley and A. Lakhotia. Identifying enumeration types modeled with symbolic constants. In L. Wills, I. Baxter, and E. Chikofsky, editors, *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 227–238. IEEE Computer Society Press, Nov. 1996.
10. HP. *DEC C Language Reference Manual*. Compaq Computer Corporation, aa-rh9na-te edition, July 1999.
11. IAR Systems. *PICmicro C Compiler: Programming Guide*, iccpic-1 edition, 1998.
12. IBM. *Developing PowerPC Embedded Application Binary (EABI) Compliant Programs*. IBM, Sept. 1998.
13. IBM. *WebSphere Development Studio ILE C/C++ Programmer's Guide*. IBM Canada Ltd, Ontario, Canada, sc09-27 12-02 edition, May 2001.
14. IBM Canada Ltd. *C for AIX Compiler Reference*. International Business Machines Corporation, May 2002.
15. Intel. *MCS 51 Microcontroller Family User's Manual*. Intel, Inc, 272383-002 edition, Feb. 1994.
16. ISO. *ISO/IEC 9945-1:1990 Information technology —Portable Operating System Interface (POSIX)*. ISO, 1990.
17. ISO. *ISO/IEC 10967-1:1994(E) Information technology —Language independent arithmetic —Part 1: Integer and floating point arithmetic*. ISO, 1994.
18. D. M. Jones. The Model C Implementation. Knowledge Software Ltd, 1992.
19. Keil. *C Compiler manual*. Keil Software, Inc, ??? edition, May 2005.
20. J. Lakos. *Large Scale C++ Software Design*. Addison–Wesley, 1996.
21. Microchip. *PIC18CXX2 High-Performance Microcontrollers with 10-Bit A/D*, ds39026b edition, 1999.
22. Motorola, Inc. *AltiVec Technology Programming Interface Manual*. Motorola, Inc, 1999.
23. H. Packard. *HP C/iX Reference Manual*. Hewlett Packardi, Inc, USA, 3 edition, Apr. 1999.
24. P. Resnick. Semantic similarity in a taxonomy: An information based measure and its application to ambiguity in natural language. *Journal of A.I. Research*, 11:95–130, 1999.
25. L. Rosler. The evolution of C–past and future. *AT&T Bell Laboratories Technical Journal*, 63(8):1685–1699, 1984.
26. J. L. Steffen. Adding run-time checking to the portable C compiler. *Software–Practice and Experience*, 22(4):305–316, 1992.
27. C.-L. Su, C.-Y. Rsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. Technical Report ACAL-TR-94-01, University of Southern California - ISI, Feb. 1994.
28. The Santa Cruz Operation. *System V Application Binary Interface: MIPS RISC processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, 3rd edition, Feb. 1996.
29. The Santa Cruz Operation. *System V Application Binary Interface: SPARC processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, third edition, 1996.
30. The Santa Cruz Operation. *System V Application Binary Interface: Intel386 Architecture Processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, fourth edition, Mar. 1997.
31. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.
32. Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In R. N. Horspool, editor, *Compiler Construction* 11^{th} *International Conference, CC 2002*, pages 14–28. Springer-Verlag, Apr. 2002.
33. S. Zucker and K. Karhi. *System V Application Binary Interface: PowerPC processor Supplement*. SunSoft, Mountain View, CA, USA, 802-3334-10 edition, Sept. 1995.