C data types

In the C programming language, **data types** constitute the semantics and characteristics of storage of data elements. They are expressed in the language syntax in form of declarations for memory locations or variables. Data types also determine the types of operations or methods of processing of data elements.

The C language provides basic arithmetic types, such as integer and real number types, AND syntax to build array and compound types. *Headers* for the C standard library, to be used via include directives, contain definitions of support types, that have additional properties, such as providing storage with an exact size, independent of the language implementation on specific hardware platforms.[1][2]

Contents

Basic types

Main types

The C language provides the four basic arithmetic type specifiers *char*, *int*, *float* and *double*, and the modifiers *signed*, *unsigned*, *short*, and *long*. The following table lists the permissible combinations in specifying a large set of storage size-specific declarations.

Type	Explanation	Minimum size (bits)	Format specifier
`char`	Smallest addressable unit of the machine that can contain basic character set. It is an integer type. Actual type can be either signed or unsigned. It contains CHAR_BIT bits.[3]	8	`%c`
`signed char`	Of the same size as *char*, but guaranteed to be signed. Capable of containing at least the [−127, +127] range.[3][a]	8	`%c` (or `%hhi` for numerical output)
`unsigned char`	Of the same size as *char*, but guaranteed to be unsigned. Contains at least the [0, 255] range.[5]	8	`%c` (or `%hhu` for numerical output)

`short`
`short int`
`signed short`
`signed short int`
	Short signed integer type. Capable of containing at least the [−32,767, +32,767] range.[3][a]	16	`%hi` or `%hd`

`unsigned short` `unsigned short int`	*Short* unsigned integer type. Contains at least the [0, 65,535] range.[3]	16	`%hu`

`int`
`signed`
`signed int`
	Basic signed integer type. Capable of containing at least the [−32,767, +32,767] range.[3][a]	16	`%i` or `%d`

`unsigned` `unsigned int`	Basic unsigned integer type. Contains at least the [0, 65,535] range.[3]	16	`%u`

`long`
`long int`
`signed long`
`signed long int`
	Long signed integer type. Capable of containing at least the [−2,147,483,647, +2,147,483,647] range.[3][a]	32	`%li` or `%ld`

`unsigned long` `unsigned long int`	*Long* unsigned integer type. Capable of containing at least the [0, 4,294,967,295] range.[3]	32	`%lu`

`long long`
`long long int`
`signed long long`
`signed long long int`
	Long long signed integer type. Capable of containing at least the [−9,223,372,036,854,775,807, +9,223,372,036,854,775,807] range.[3][a] Specified since the C99 version of the standard.	64	`%lli` or `%lld`

`unsigned long long` `unsigned long long int`	*Long long* unsigned integer type. Contains at least the [0, +18,446,744,073,709,551,615] range.[3] Specified since the C99 version of the standard.	64	`%llu`

`float`	Real floating-point type, usually referred to as a single-precision floating-point type. Actual properties unspecified (except minimum limits); however, on most systems, this is the IEEE 754 single-precision binary floating-point format (32 bits). This format is required by the optional Annex F "IEC 60559 floating-point arithmetic".		Converting from text:[b]
			`%f %F`
			`%g %G`
			`%e %E`
			`%a %A`

`double`	Real floating-point type, usually referred to as a double-precision floating-point type. Actual properties unspecified (except minimum limits); however, on most systems, this is the IEEE 754 double-precision binary floating-point format (64 bits). This format is required by the optional Annex F "IEC 60559 floating-point arithmetic".		`%lf %lF`
			`%lg %lG`
			`%le %lE`
			`%la %lA`[c]

`long double`	Real floating-point type, usually mapped to an extended precision floating-point number format. Actual properties unspecified. It can be either x86 extended-precision floating-point format (80 bits, but typically 96 bits or 128 bits in memory with padding bytes), the non-IEEE "double-double" (128 bits), IEEE 754 quadruple-precision floating-point format (128 bits), or the same as double. See the article on long double for details.		`%Lf %LF`
			`%Lg %LG`
			`%Le %LE`
			`%La %LA`[c]

a. The minimal ranges $-(2^{n-1}-1)$ to $2^{n-1}-1$ (e.g. [−127,127]) come from the various integer representations allowed by the standard (ones' complement, sign-magnitude, two's complement).[4] However, most platforms use two's complement, implying a range of the form -2^{m-1} to $2^{m-1}-1$ with m ≥ n for these implementations, e.g. [−128,127] (SCHAR_MIN = −128 and SCHAR_MAX = 127) for an 8-bit *signed char*.

b. These format strings also exist for formatting to text, but operate on a double.
c. Uppercase differs from lowercase in the output. Uppercase specifiers produce values in the uppercase, and lowercase in lower (%A, %E, %F, %G produce such values as INF, NAN and E (exponent) in uppercase)

The actual size of the integer types varies by implementation. The standard requires only size relations between the data types and minimum sizes for each data type:

The relation requirements are that the `long long` is not smaller than `long`, which is not smaller than `int`, which is not smaller than `short`. As `char`'s size is always the minimum supported data type, no other data types (except bit-fields) can be smaller.

The minimum size for `char` is 8 bits, the minimum size for `short` and `int` is 16 bits, for `long` it is 32 bits and `long long` must contain at least 64 bits.

The type `int` should be the integer type that the target processor is most efficiently working with. This allows great flexibility: for example, all types can be 64-bit. However, several different integer width schemes (data models) are popular. Because the data model defines how different programs communicate, a uniform data model is used within a given operating system application interface.[6]

In practice, `char` is usually 8 bits in size and `short` is usually 16 bits in size (as are their unsigned counterparts). This holds true for platforms as diverse as 1990s SunOS 4 Unix, Microsoft MS-DOS, modern Linux, and Microchip MCC18 for embedded 8-bit PIC microcontrollers. POSIX requires `char` to be exactly 8 bits in size.

Various rules in the C standard make `unsigned char` the basic type used for arrays suitable to store arbitrary non-bit-field objects: its lack of padding bits and trap representations, the definition of *object representation*,[5] and the possibility of aliasing.[7]

The actual size and behavior of floating-point types also vary by implementation. The only guarantee is that `long double` is not smaller than `double`, which is not smaller than `float`. Usually, the 32-bit and 64-bit IEEE 754 binary floating-point formats are used.

The C99 standard includes new real floating-point types `float_t` and `double_t`, defined in `<math.h>`. They correspond to the types used for the intermediate results of floating-point expressions when `FLT_EVAL_METHOD` is 0, 1, or 2. These types may be wider than `long double`.

C99 also added complex types: `float _Complex`, `double _Complex`, `long double _Complex`.

Boolean type

C99 added a boolean (true/false) type `_Bool`. Additionally, the `<stdbool.h>` header defines `bool` as a convenient alias for this type, and also provides macros for `true` and `false`. `_Bool` functions similarly to a normal integer type, with one exception: any assignments to a `_Bool` that are not 0 (false) are stored as 1 (true). This behavior exists to avoid integer overflows in implicit narrowing conversions. For example, in the following code:

```
unsigned char b = 256;

if (b) {
    /* do something */
}
```

Variable `b` evaluates to false if `unsigned char` has a size of 8 bits. This is because the value 256 does not fit in the data type, which results in the lower 8 bits of it being used, resulting in a zero value. However, changing the type causes the previous code to behave normally:

```
_Bool b = 256;

if (b) {
```

```
    /* do something */
}
```

The type *_Bool* also ensures true values always compare equal to each other:

```
_Bool a = 1, b = 2;

if (a == b) {
    /* do something */
}
```

Size and pointer difference types

The C language specification includes the `typedefs` `size_t` and `ptrdiff_t` to represent memory-related quantities. Their size is defined according to the target processor's arithmetic capabilities, not the memory capabilities, such as available address space. Both of these types are defined in the `<stddef.h>` header (`cstddef` in C++).

`size_t` is an unsigned integer type used to represent the size of any object (including arrays) in the particular implementation. The operator `sizeof` yields a value of the type `size_t`. The maximum size of `size_t` is provided via `SIZE_MAX`, a macro constant which is defined in the `<stdint.h>` header (`cstdint` header in C++). `size_t` is guaranteed to be at least 16 bits wide. Additionally, POSIX includes `ssize_t`, which is a signed integer type of the same width as `size_t`.

`ptrdiff_t` is a signed integer type used to represent the difference between pointers. It is guaranteed to be valid only against pointers of the same type; subtraction of pointers consisting of different types is implementation-defined.

Interface to the properties of the basic types

Information about the actual properties, such as size, of the basic arithmetic types, is provided via macro constants in two headers: `<limits.h>` header (`climits` header in C++) defines macros for integer types and `<float.h>` header (`cfloat` header in C++) defines macros for floating-point types. The actual values depend on the implementation.

Properties of integer types

- `CHAR_BIT` – size of the char type in bits (at least 8 bits)
- `SCHAR_MIN`, `SHRT_MIN`, `INT_MIN`, `LONG_MIN`, `LLONG_MIN`(C99) – minimum possible value of signed integer types: signed char, signed short, signed int, signed long, signed long long
- `SCHAR_MAX`, `SHRT_MAX`, `INT_MAX`, `LONG_MAX`, `LLONG_MAX`(C99) – maximum possible value of signed integer types: signed char, signed short, signed int, signed long, signed long long
- `UCHAR_MAX`, `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX`, `ULLONG_MAX`(C99) – maximum possible value of unsigned integer types: unsigned char, unsigned short, unsigned int, unsigned long, unsigned long long
- `CHAR_MIN` – minimum possible value of char
- `CHAR_MAX` – maximum possible value of char
- `MB_LEN_MAX` – maximum number of bytes in a multibyte character

Properties of floating-point types

- `FLT_MIN`, `DBL_MIN`, `LDBL_MIN` – minimum normalized positive value of float, double, long double respectively
- `FLT_TRUE_MIN`, `DBL_TRUE_MIN`, `LDBL_TRUE_MIN` (C11) – minimum positive value of float, double, long double respectively
- `FLT_MAX`, `DBL_MAX`, `LDBL_MAX` – maximum finite value of float, double, long double, respectively
- `FLT_ROUNDS` – rounding mode for floating-point operations

- FLT_EVAL_METHOD (C99) – evaluation method of expressions involving different floating-point types
- FLT_RADIX – radix of the exponent in the floating-point types
- FLT_DIG, DBL_DIG, LDBL_DIG – number of decimal digits that can be represented without losing precision by float, double, long double, respectively
- FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON – difference between 1.0 and the next representable value of float, double, long double, respectively
- FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG – number of FLT_RADIX-base digits in the floating-point significand for types float, double, long double, respectively
- FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP – minimum negative integer such that FLT_RADIX raised to a power one less than that number is a normalized float, double, long double, respectively
- FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP – minimum negative integer such that 10 raised to that power is a normalized float, double, long double, respectively
- FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP – maximum positive integer such that FLT_RADIX raised to a power one less than that number is a normalized float, double, long double, respectively
- FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP – maximum positive integer such that 10 raised to that power is a normalized float, double, long double, respectively
- DECIMAL_DIG (C99) – minimum number of decimal digits such that any number of the widest supported floating-point type can be represented in decimal with a precision of DECIMAL_DIG digits and read back in the original floating-point type without changing its value. DECIMAL_DIG is at least 10.

Fixed-width integer types

The C99 standard includes definitions of several new integer types to enhance the portability of programs.[2] The already available basic integer types were deemed insufficient, because their actual sizes are implementation defined and may vary across different systems. The new types are especially useful in embedded environments where hardware usually supports only several types and that support varies between different environments. All new types are defined in `<inttypes.h>` header (`cinttypes` header in C++) and also are available at `<stdint.h>` header (`cstdint` header in C++). The types can be grouped into the following categories:

- Exact-width integer types that are guaranteed to have the same number *n* of bits across all implementations. Included only if it is available in the implementation.
- Least-width integer types that are guaranteed to be the smallest type available in the implementation, that has at least specified number *n* of bits. Guaranteed to be specified for at least N=8,16,32,64.
- Fastest integer types that are guaranteed to be the fastest integer type available in the implementation, that has at least specified number *n* of bits. Guaranteed to be specified for at least N=8,16,32,64.
- Pointer integer types that are guaranteed to be able to hold a pointer. Included only if it is available in the implementation.
- Maximum-width integer types that are guaranteed to be the largest integer type in the implementation.

The following table summarizes the types and the interface to acquire the implementation details (*n* refers to the number of bits):

Type category	Signed types			Unsigned types		
	Type	Minimum value	Maximum value	Type	Minimum value	Maximum value
Exact width	int*n*_t	INT*n*_MIN	INT*n*_MAX	uint*n*_t	0	UINT*n*_MAX
Least width	int_least*n*_t	INT_LEAST*n*_MIN	INT_LEAST*n*_MAX	uint_least*n*_t	0	UINT_LEAST*n*_MAX
Fastest	int_fast*n*_t	INT_FAST*n*_MIN	INT_FAST*n*_MAX	uint_fast*n*_t	0	UINT_FAST*n*_MAX
Pointer	intptr_t	INTPTR_MIN	INTPTR_MAX	uintptr_t	0	UINTPTR_MAX
Maximum width	intmax_t	INTMAX_MIN	INTMAX_MAX	uintmax_t	0	UINTMAX_MAX

Printf and scanf format specifiers

The <inttypes.h> header (cinttypes in C++) provides features that enhance the functionality of the types defined in the <stdint.h> header. It defines macros for printf format string and scanf format string specifiers corresponding to the types defined in <stdint.h> and several functions for working with the intmax_t and uintmax_t types. This header was added in C99.

Printf format string

The macros are in the format PRI*{fmt}{type}*. Here *{fmt}* defines the output formatting and is one of d (decimal), x (hexadecimal), o (octal), u (unsigned) and i (integer). *{type}* defines the type of the argument and is one of *n*, FAST*n*, LEAST*n*, PTR, MAX, where *n* corresponds to the number of bits in the argument.

Scanf format string

The macros are in the format SCN*{fmt}{type}*. Here *{fmt}* defines the output formatting and is one of d (decimal), x (hexadecimal), o (octal), u (unsigned) and i (integer). *{type}* defines the type of the argument and is one of *n*, FAST*n*, LEAST*n*, PTR, MAX, where *n* corresponds to the number of bits in the argument.

Functions

Additional floating-point types

Similarly to the fixed-width integer types, ISO/IEC TS 18661 specifies floating-point types for IEEE 754 interchange and extended formats in binary and decimal:

- _Float**N** for binary interchange formats;
- _Decimal**N** for decimal interchange formats;
- _Float**N**x for binary extended formats;
- _Decimal**N**x for decimal extended formats.

Structures

Structures aggregate the storage of multiple data items, of potentially differing data types, into one memory block referenced by a single variable. The following example declares the data type struct birthday which contains the name and birthday of a person. The structure definition is followed by a declaration of the variable John that allocates the needed storage.

```
struct birthday {
    char name[20];
    int day;
    int month;
    int year;
};

struct birthday John;
```

The memory layout of a structure is a language implementation issue for each platform, with a few restrictions. The memory address of the first member must be the same as the address of structure itself. Structures may be initialized or assigned to using compound literals. A function may directly return a structure, although this is often not efficient at run-time. Since C99, a structure may also end with a flexible array member.

A structure containing a pointer to a structure of its own type is commonly used to build linked data structures:

```
struct node {
    int val;
    struct node *next;
};
```

Arrays

For every type `T`, except void and function types, there exist the types *"array of `N` elements of type `T`"*. An array is a collection of values, all of the same type, stored contiguously in memory. An array of size `N` is indexed by integers from `0` up to and including `N−1`. Here is a brief example:

```
int cat[10];  // array of 10 elements, each of type int
```

Arrays can be initialized with a compound initializer, but not assigned. Arrays are passed to functions by passing a pointer to the first element. Multidimensional arrays are defined as *"array of array …"*, and all except the outermost dimension must have compile-time constant size:

```
int a[10][8];  // array of 10 elements, each of type 'array of 8 int elements'
```

Pointers

Every data type `T` has a corresponding type *pointer to `T`*. A pointer is a data type that contains the address of a storage location of a variable of a particular type. They are declared with the asterisk (`*`) type declarator following the basic storage type and preceding the variable name. Whitespace before or after the asterisk is optional.

```
char *square;
long *circle;
int *oval;
```

Pointers may also be declared for pointer data types, thus creating multiple indirect pointers, such as `char **` and `int ***`, including pointers to array types. The latter are less common than an array of pointers, and their syntax may be confusing:

```
char *pc[10];   // array of 10 elements of 'pointer to char'
char (*pa)[10]; // pointer to a 10-element array of char
```

The element `pc` requires ten blocks of memory of the size of *pointer to `char`* (usually 40 or 80 bytes on common platforms), but element `pa` is only one pointer (size 4 or 8 bytes), and the data it refers to is an array of ten bytes (`sizeof *pa == 10`).

Unions

A union type is a special construct that permits access to the same memory block by using a choice of differing type descriptions. For example, a union of data types may be declared to permit reading the same data either as an integer, a float, or any other user declared type:

```
union {
    int i;
    float f;
    struct {
        unsigned int u;
        double d;
    } s;
} u;
```

The total size of `u` is the size of `u.s` – which happens to be the sum of the sizes of `u.s.u` and `u.s.d` – since `s` is larger than both `i` and `f`. When assigning something to `u.i`, some parts of `u.f` may be preserved if `u.i` is smaller than `u.f`.

Reading from a union member is not the same as casting since the value of the member is not converted, but merely read.

Function pointers

Function pointers allow referencing functions with a particular signature. For example, to store the address of the standard function `abs` in the variable `my_int_f`:

```
int (*my_int_f)(int) = &abs;
// the & operator can be omitted, but makes clear that the "address of" abs is used here
```

Function pointers are invoked by name just like normal function calls. Function pointers are separate from pointers and void pointers.

Type qualifiers

The aforementioned types can be characterized further by type qualifiers, yielding a *qualified type*. As of 2014 and C11, there are four type qualifiers in standard C: `const` (C89), `volatile` (C89), `restrict` (C99) and `_Atomic` (C11) – the latter has a private name to avoid clashing with user names,[8] but the more ordinary name `atomic` can be used if the `<stdatomic.h>` header is included. Of these, `const` is by far the best-known and most used, appearing in the standard library and encountered in any significant use of the C language, which must satisfy const-correctness. The other qualifiers are used for low-level programming, and while widely used there, are rarely used by typical programmers.

See also

- C syntax
- Uninitialized variable
- Integer (computer science)

References

1. Barr, Michael (2 December 2007). "Portable Fixed-Width Integers in C" (https://barrgroup.com/Embedded-Systems/How-To/C-Fixed-Width-Integers-C99). Retrieved 18 January 2016.
2. *ISO/IEC 9899:1999 specification, TC3* (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf) (PDF). p. 255, § 7.18 *Integer types <stdint.h>*.
3. *ISO/IEC 9899:1999 specification, TC3* (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf) (PDF). p. 22, § 5.2.4.2.1 *Sizes of integer types <limits.h>*.
4. *Rationale for International Standard—Programming Languages—C Revision 5.10* (http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf) (PDF). p. 25, § 5.2.4.2.1 *Sizes of integer types <limits.h>*.
5. *ISO/IEC 9899:1999 specification, TC3* (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf) (PDF). p. 37, § 6.2.6.1 *Representations of types – General*.
6. "64-Bit Programming Models: Why LP64?" (http://www.unix.org/version2/whatsnew/lp64_wp.html). The Open Group. Retrieved 9 November 2011.
7. *ISO/IEC 9899:1999 specification, TC3* (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf) (PDF). p. 67, § 6.5 *Expressions*.
8. C11:The New C Standard (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3631.pdf), Thomas Plum

organization.