C data types

From Wikipedia, the free encyclopedia

In the C programming language, **data types** are declarations for memory locations or variables that determine the characteristics of the data that may be stored and the methods (operations) of processing that are permitted involving them.

The C language provides basic arithmetic types, such as integer and real number types, and syntax to build array and compound types. Several headers in the C standard library contain definitions of support types, that have additional properties, such as providing storage with an exact size, independent of the implementation.[1][2]

Contents

- 1 Basic types
  - 1.1 Boolean type
  - 1.2 Size and pointer difference types
  - 1.3 Interface to the properties of the basic types
- 2 Fixed-width integer types
  - 2.1 Printf and scanf format specifiers
- 3 Structures
- 4 Arrays
- 5 Pointer types
- 6 Unions
- 7 Function pointers
- 8 Type qualifiers
- 9 See also
- 10 References

Basic 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 to specify a large set of storage size-specific declarations.

Type	Explanation	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 depending on the implementation. It contains `CHAR_BIT` bits.[3]	%c
signed char	Of the same size as `char`, but guaranteed to be signed. Capable of containing **at least** the [−127, +127] range;[3][4]	%c (or %hhi for numerical output)
unsigned char	Of the same size as `char`, but guaranteed to be unsigned. It is represented in binary notation without padding bits; thus, its range is exactly [0, $2^{\texttt{CHAR\_BIT}} - 1$].[5]	%c (or %hhu for numerical output)
short short int signed short signed short int	*Short* signed integer type. Capable of containing **at least** the [−32767, +32767] range;[3][4] thus, it is at least 16 bits in size. The negative value is −32767 (not −32768) due to the one's-complement and sign-magnitude representations allowed by the standard, though the two's-complement representation is much more common.[6]	%hi
unsigned short unsigned short int	Similar to `short`, but unsigned.	%hu
int signed signed int	Basic signed integer type. Capable of containing **at least** the [−32767, +32767] range;[3][4] thus, it is at least 16 bits in size.	%i or %d
unsigned unsigned int	Similar to `int`, but unsigned.	%u
long long int signed long signed long int	*Long* signed integer type. Capable of containing **at least** the [−2147483647, +2147483647] range;[3][4] thus, it is at least 32 bits in size.	%li
unsigned long unsigned long int	Similar to `long`, but unsigned.	%lu
long long long long int signed long long signed long long int	*Long long* signed integer type. Capable of containing **at least** the [−9223372036854775807, +9223372036854775807] range;[3][4] thus, it is at least 64 bits in size. Specified since the C99 version of the standard.	%lli
unsigned long long unsigned long long int	Similar to `long long`, but unsigned. Specified since the C99 version of the standard.	%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. This format is required by the optional Annex F "IEC 60559 floating-point arithmetic".	%f (promoted automatically to `double` for `printf()`)
	Real floating-point type, usually referred to as a double-	%f (%F)

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. This format is required by the optional Annex F "IEC 60559 floating-point arithmetic".	(%lf (%lF) for `scanf()`) %g %G %e %E (for scientific notation)[7]
long double	Real floating-point type, usually mapped to an extended precision floating-point number format. Actual properties unspecified. Unlike types float and double, it can be either 80-bit floating point format, the non-IEEE "double-double" or IEEE 754 quadruple-precision floating-point format if a higher precision format is provided, otherwise it is the same as double. See the article on long double for details.	%Lf %LF %Lg %LG %Le %LE[7]

The actual size of integer types varies by implementation. The standard only requires 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. This is because the data model defines how different programs communicate, a uniform data model is used within a given operating system application interface.[8]

In practice, `char` is usually eight 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 eight 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.[9]

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, if supported by hardware.

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 new `<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 integral 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 */
}
```

`b` evaluates to false if `unsigned char` is 8 bits wide. This is because 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 */
}
```

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` header in C++).

`size_t` is an unsigned integer type used to represent the size of any object (including arrays) in the particular implementation. The `sizeof` operator 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 integral type of the same width as `size_t`.

`ptrdiff_t` is a signed integral type used to represent the difference between pointers. It is only guaranteed to be valid 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 which 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 which 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 which 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 which are guaranteed to be able to hold a pointer. Included only if it is available in the implementation.
- Maximum-width integer types which 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	`intN_t`	`INTN_MIN`	`INTN_MAX`	`uintN_t`	0	`UINTN_MAX`
Least width	`int_leastN_t`	`INT_LEASTN_MIN`	`INT_LEASTN_MAX`	`uint_leastN_t`	0	`UINT_LEASTN_MAX`
Fastest	`int_fastN_t`	`INT_FASTN_MIN`	`INT_FASTN_MAX`	`uint_fastN_t`	0	`UINT_FASTN_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` header in C++) provides features that enhance the functionality of the types defined in `<stdint.h>` header. Included are macros that define printf format string and scanf format string specifiers corresponding to the `<stdint.h>` types and several functions for working with `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`, `FASTN`, `LEASTN`, `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`, `FASTN`, `LEASTN`, `PTR`, `MAX`, where `N` corresponds to the number of bits in the argument.

Functions

Structures

Structures are a way of storing multiple pieces of data in one variable. For example, say we wanted to store the name and birthday of a person in strings, in one variable. We could use a structure to house that data:

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

Structures may contain pointers to structs of its own type, which is common in linked data structures.

A C implementation has freedom to design the memory layout of the struct, with few restrictions; one being that the memory address of the first member will be the same as the address of struct itself. Structs may be initialized or assigned to using compound literals. A user-written function can directly return a structure, though it will often not be very efficient at run-time. Since C99, a struct can also end with a flexible array member.

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*. There are also "arrays of unspecified size" where the number of elements is not known by the compiler. Here is a brief example:

```
int cat[10];  // array of 10 elements, each of type int
int bob[];    // array of an unspecified number of 'int' elements
```

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'
float f[][32]; // array of unspecified number of 'array of 32 float elements'
```

Pointer types

For every type *T* there exists a type “pointer to *T*”.

Variables can be declared as being pointers to values of various types, by means of the `*` type declarator. To declare a variable as a pointer, precede its name with an asterisk.

```
char *square;
long *circle;
```

Hence "for every type T" also applies to pointer types there exists multi-indirect pointers like `char**` or `int***` and so on. There exists also "pointer to array" types, but they are less common than "array of pointer", and their syntax is quite confusing:

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

`pc` consumes 10×`sizeof(char*)` bytes (usually 40 or 80 bytes on common platforms), but `pa` is only one pointer, so `sizeof(pa)` is usually 4 or 8, and the data it refers to is an array of 10 bytes: `sizeof(*pa) == 10`.

Unions

Union types are special structures which allow access to the same memory using different type descriptions; one could, for example, describe a union of data types which would allow reading the same data as an integer, a float or a user declared type:

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

In the above example 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,[10] 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)

The Wikibook *C Programming* has a page on the topic of: ***C Programming***

References

1. Barr, Michael (2 December 2007). "Portable Fixed-Width Integers in C". Retrieved 18 January 2016.
2. *ISO/IEC 9899:1999 specification, TC3* (PDF). p. 255, § 7.18 *Integer types* `<stdint.h>`.
3. *ISO/IEC 9899:1999 specification, TC3* (PDF). p. 22, § 5.2.4.2.1 *Sizes of integer types* `<limits.h>`.
4. Despite the intervals of **$-(2^{n-1}-1)$** to **$2^{n-1}-1$** laid out in the standard, most compilers (gcc, clang with all warning flags, and msvc) use integral ranges of **-2^{n-1}** to **$2^{n-1}-1$**, e.g. [-128,127] (SCHAR_MIN = -128 and SCHAR_MAX = 127 for INT8_t)
5. *ISO/IEC 9899:1999 specification, TC3* (PDF). p. 37, § 6.2.6.1 *Representations of types — General.*
6. *Rationale for International Standard—Programming Languages—C Revision 5.10* (PDF). p. 25, § 5.2.4.2.1 *Sizes of integer types <limits.h>*.

7. Uppercase differs from lowercase in the output. Uppercase specifiers produce values in the uppercase, and lowecase in lower (%E, %F, %G produce such values as INF, NAN and E (exponent) in uppercase).
8. "64-Bit Programming Models: Why LP64?". The Open Group. Retrieved 9 November 2011.
9. *ISO/IEC 9899:1999 specification, TC3* (PDF). p. 67, § 6.5 *Expressions*.
10. C11:The New C Standard (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3631.pdf), Thomas Plum