

# I

**I<sup>C99</sup>** A macro, defined in `complex.h`, that expands to `_Complex_I` if the macro `_Imaginary_I` is not defined; otherwise, it expands to `_Imaginary_I`.

**I18N** An abbreviation for internationalization, a word that has 18 letters between its first and last letter. *See also* L10N.

**identifier** A name consisting of a sequence of characters that can be used to name a variable, function, enumeration constant, type synonym, structure, union or enumerated type tag or member, label, or macro. An identifier must begin with a letter or an underscore and may contain alphanumeric and underscore characters.

While K&R stated that the first eight characters were significant, C89 requires that identifiers be significant in at least the first 31 (C99 requires 63) characters and be case distinct. (Note that due to linker and other restrictions, the length of significance of external names may be as few as 6 in C89 and 31 in C99. Prior to C99, they also may be converted to one case only.)

As a rule, to avoid conflicts between user-written and vendor-supplied identifiers, do not invent an identifier whose name begins with an underscore. (While the requirements are not quite this strict, this rule is much simpler to remember.)

Two identifiers may designate different entities within the same scope provided they are in different name spaces.

**identifier conflicts with C++** Conflicts arising when C++-only keywords are used in a C program and that program is compiled in a C++ environment. To avoid such conflicts, you should refrain from using the following as identifiers in new C code you write: `asm`, `catch`, `class`, `const_cast`, `delete`, `dynamic_cast`, `explicit`, `export`, `friend`, `mutable`, `namespace`, `new`, `operator`, `private`, `protected`, `public`, `reinterpret_cast`, `static_cast`, `template`, `this`, `throw`, `try`, `typeid`, `typename`, `using`, and `virtual`.

C++ also has the keywords `bool`, `false`, and `true`, which while they are not keywords in Standard C, they exist in C99 as macro names defined in the header `stdbool.h`. C99 added the keyword `inline`, which C++ already had. C89 defined `wchar_t` as a type synonym while C++ makes this a keyword.

**identifier linkage** *See* linkage.

**identifier list** A comma-separated list of identifiers.

**identifier name space** *See* name space.

**identifier, reserved** Keywords that are special identifiers in that they are reserved words. As such, they cannot be used as user-defined identifiers. Also, all external names declared in the standard headers are reserved *whether or not* their parent header is actually included. *See also* future library directions.

**identifier scope** *See* scope.

**identifier type** *See* type.

**IEC** The International Electrotechnical Commission, a body that develops standards. *See also* ISO.

**IEC 559** *See* IEEE Floating-Point Arithmetic Standards.

**IEC 60559** *See* IEEE Floating-Point Arithmetic Standards.

**IEEE** The Institute of Electrical and Electronics Engineers, a U.S.-based organization that is involved, among other things, in producing various computer-related hardware and software standards and specifications.

**IEEE 754** *See* IEEE Floating-Point Arithmetic Standards.

**IEEE 854** *See* IEEE Floating-Point Arithmetic Standards.

**IEEE Floating-Point Arithmetic Standards** IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) known internationally first as IEC 559:1989 (and then as IEC 60559:1989) and IEEE Standard for Radix-Independent Floating-Point Arithmetic (ANSI/IEEE Std 854-1987). C99 added extensive support for floating-point environments adhering to these standards.

**#if** A preprocessor directive used to begin a conditional compilation “block” based on the truth of an integer constant expression. It must have a corresponding **#endif** directive and, optionally, **#else** or **#elif** directives. It is used as follows:

**#if** *constant-expression*

If *constant-expression* contains any identifiers that are not currently defined as macros, they are assumed (for this directive only) to be macros defined with value 0. *constant-expression* is evaluated as though each term had type `long int`. In C99, all signed integer types and all unsigned integer types act like the types `intmax_t` and `uintmax_t`, respectively. *See also* comment.

**#ifdef** A preprocessor directive used to begin a conditional compilation “block” based on the existence of a macro definition. It must have a

corresponding `#endif` directive and, optionally, a `#else` or `#elif` directive. It is used as follows:

```
#ifdef identifier
```

`if/else` A construct involving these two keywords that allows control to be directed based on the truth value of a scalar expression. The true and false paths must consist of only one statement each. If more than one is needed, a block must be used. It is used as follows:

```
if ( expression )
    statement
[ else
    statement ]
```

Note that *expression* is a full expression. If it contains no relational or equality operator, inequality to zero is implied; for example, `if (p)` is equivalent to `if (p != 0)`.

`#ifndef` A preprocessor directive used to begin a conditional compilation “block” based on the nonexistence of a macro definition. It must have a corresponding `#endif` directive and, optionally, a `#else` or `#elif` directive. It is used as follows:

```
#ifndef identifier
```

`ilogb[f|l]`<sup>C99</sup> A function that returns the exponent of *x*.

```
#include <math.h>
int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

If *x* is zero, `FP_ILOGB0` is returned. If *x* is infinite, `INT_MAX` is returned. If *x* is a NaN, `FP_ILOGBNAN` is returned. If *x* is zero, a range error may result.

`imaginary`<sup>C99</sup> A macro, defined in `complex.h`, that is defined only if the implementation supports imaginary types according to the standard’s Annex G. It expands to the keyword `_Imaginary`. For normal usage of the imaginary type, it is strongly recommended that you include `complex.h` and use this macro rather than using the `_Imaginary` keyword directly. There are three imaginary types: `float _Imaginary`, `double _Imaginary`, and `long double _Imaginary`, which can be written instead as `float imaginary`, `double imaginary`, and `long double imaginary`, respectively.

**\_Imaginary**<sup>C99</sup> A keyword that provides support for an imaginary type. Since it was invented by C99, by which time quite a few programs already contained type synonyms using some form of the word *complex*, this keyword was spelled using one of the forms reserved for implementers. Unless you must mix both existing homegrown complex machinery and this new keyword in the same source file, it is strongly recommended that you include `complex.h` and use its macro `imaginary` instead.

Note that implementations are not required to provide imaginary types.

**\_Imaginary\_I**<sup>C99</sup> A macro, defined in `complex.h`, that is defined only if the implementation supports imaginary types according to the standard's Annex G. It expands to a constant expression of type `const float` **\_Imaginary**, having a value of the imaginary unit *i*.

**imaxabs**<sup>C99</sup> A function that computes the absolute value of its argument.

```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
```

**imaxdiv**<sup>C99</sup> A function that computes `numer/denom` and `numer % denom` in a single operation.

```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

The quotient and the remainder are returned in a structure of type `imaxdiv_t`.

**imaxdiv\_t**<sup>C99</sup> A type, defined in `inttypes.h`, that is a structure that contains (in either order) the members `quot` (representing a quotient) and `rem` (representing a remainder), each of which has type `intmax_t`. *See also* `imaxdiv`.

**implementation** A C translation environment. It collectively refers to the preprocessor, compiler, and run-time library for a given host system. It is used primarily to mean “compiler” or “interpreter.”

**implementation, conforming** A C translation environment that conforms to the standard specification for C. Specifically, a hosted implementation that conforms to C89 must contain all of the library functions, whereas the set of library facilities provided by a conforming freestanding implementation is implementation defined, beyond having to provide `float.h`, `limits.h`, `stdarg.h`, and `stddef.h`. C99 requires three extra freestanding headers: `iso646.h`, `stdbool.h`, and `stdint.h`.

*See also* `__STDC__`; `__STDC_VERSION__`.

**implementation limits** *See* environmental limits.

**implementation-defined behavior** Given a correct program construct and correct data, if the runtime behavior depends on the characteristics of the implementation, then that behavior is deemed to be implementation defined. Such behavior shall be documented by that implementation. For example, whether or not a `char` is sign extended on conversion to `int` is implementation defined, as is whether a plain `int` bit-field is signed.

**implicit conversion** *See* conversion; conversion, explicit; conversion, implicit.

**implicit function declaration** The implicit process that occurs if the compiler comes across an identifier for which it has no visible declaration in scope, and that identifier is the expression that immediately precedes a function call operator. The identifier is implicitly declared as follows:

```
extern int identifier();
```

Note that no prototype information is assumed. Consider the following example:

```
void a(void)
{
    void f(void);

    f();
}

void b(void)
{
    int i;

    i = f(10); /* discrepancy not detected */
}
```

Because the declaration of function `f` only has block scope, the compiler forgets about it at the end of its parent block. As such, no prototype is in scope of the second call to `f`, and the compiler is not able to detect the argument list or return type mismatches. (This can be avoided by placing the function declaration at file scope.)

**Note that C99 does not support implicit function declarations.**

**implicit int** The assumption of type `int`, when no type keyword exists in a declaration. Prior to C99, if a type were omitted in numerous

contexts, type `int` was assumed. Such occurrences of “implicit `int`” are not permitted by C99.

**#include** A preprocessor directive used to include a standard or programmer-defined header. It is used as follows:

```
#include <preprocessor-tokens>
#include "preprocessor-tokens"
#include identifier
```

where *preprocessor-tokens* is either the name of a header or a series of tokens that after preprocessor expansion, together form the name of a header.

The third case was an invention of C89 and is permitted, provided the identifier is a macro that expands to a character sequence of the form specified in either of the first two cases. *See also* header, name of.

**inclusive OR assignment operator** *See* OR assignment operator, bitwise inclusive.

**inclusive OR operator** *See* OR operator, bitwise inclusive.

**incomplete type** *See* type, incomplete.

**increment operator** A unary operator, `++`, that may be used as either a prefix or postfix operator. The operand must have a scalar type and be a modifiable lvalue. The value of `x++` is the value of `x` before it is incremented by 1, whereas the value of `++x` is the value of `x` after it is incremented by 1. The postfix version of this operator associates left to right, while the prefix version associates right to left. (Prior to C89, the prefix and postfix versions had the same precedence. However, C89 elevated the precedence of the postfix version. This broke no correct existing code. It did, however, permit previously invalid constructs to be valid. For example, `p++->m` is now valid.) There is a corresponding decrement operator `--`.

**indirection** The act of getting at an object or function via a pointer to that object or function. It is achieved using the unary `*` indirection operator.

**indirection operator** A unary operator, `*`, that is used to get at an object or function indirectly via a pointer to it. Indirection on a pointer is often called dereferencing. This operator associates right to left. When applied to data pointers, it always produces an lvalue. Given the following declarations:

```
double d;
double *pd = &d;
double **ppd = &pd;

double df(void);
double (*pf)(void) = df;
```

the expression `*pd` designates the `double` object to which `pd` points; the expression `**pd` designates the `double` object to which `*pd` points; and the expression `*pf` designates the function to which `pf` points.

**inequality operator** A binary operator, `!=`, that compares the values of its operands for inequality. Its operands must have scalar types and must be either both arithmetic, both pointers to qualified or unqualified versions of compatible types, one a pointer to `void` and the other a pointer to an object or incomplete type, or one a pointer and the other the null pointer constant. The order of evaluation of the operands is unspecified. The result has type `int` and value 0 (if false) or 1 (if true). This operator associates left to right.

**INFINITY**<sup>C99</sup> A macro, defined in `math.h`, that expands to a constant expression of type `float` that represents the value positive or unsigned infinity, if an infinity value is available. Otherwise, it expands to a positive constant of type `float` that overflows at compile time, thus causing a diagnostic.

**initialization** The act of defining an object's initial value. Objects that have automatic storage duration and are not explicitly initialized have indeterminate values. Objects that have static storage duration and are not explicitly initialized have a value of zero cast to their type. (In the case of aggregate objects, the value zero is cast to all subordinate scalar members; for unions, it is cast to the first member.) C89 added the ability to explicitly initialize unions and automatic aggregates; however, such initializers must be translation-time constant expressions. C99 added the ability to initialize given elements of an array, and given members of structures and unions via designated initializers. *See also* initializer.

**initializer** A construct of the following format used in initialization:

```
{ initializer-list }
```

If the object being initialized has scalar type, the delimiting `{` and `}` may be omitted (and usually are). *initializer-list* is a comma-separated list of initializers. Nested initializers are used to initialize nested structures, arrays of structures, multidimensional arrays, and the like. *See also* compound literal.

**initializer, string literal** An initializer for an array of `char`. There of two ways of writing such an initializer, as follows:

```
char c1[] = {'o', 'n', 'e', '\0'};
char c2[] = "one";
```

In the case of `c2`, the initializer has the form of a string literal. The initializer expression has type “array of 4 `char`” as you would expect. However, this array type is not converted to the address of the first element. Instead, it is recognized as an abbreviation for the first case. Note, though, that in the following case, the array consists of only three characters—there is no trailing `'\0'` placed in `c3`. The following is an example:

```
char c3[3] = "one";
```

**inline**<sup>C99</sup> A type specifier used on a function which suggests to the translator that calls to that function should be as fast as possible; that is, the function should be “inlined.” However, whether or not this suggestion is followed, is implementation defined. `inline` is also a C++ keyword.

**int** An integer type keyword. Standard C requires it to represent at least 16 bits. Typically, it maps onto the native data type for a given machine’s architecture (its word or register size, for example). Except when used with a bit-field, a plain `int` is signed. *See also* integer type.

**int\_curr\_symbol**<sup>C89</sup> An `lconv` structure member that is a pointer to a string containing the international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217 Codes for the Representation of Currency and Funds*. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity. If the string consists of "", this indicates that the value is not available in the current locale or is of zero length. In the "C" locale this member must have the value "". Examples are “CHF␣” (Switzerland, franc), “ITL␣” (Italy, lira), “NOK␣” (Norway, krona), and “NLG␣” (Netherlands, guilder), where ␣ represents a space.

**integer arithmetic functions** The `stdlib.h` functions `abs`, `div`, `labs`, `ldiv`, `llabs` and `lldiv`.

**integer constant** *See* constant, integer.

**integer constant expression** An expression involving only integer constants, enumeration constants, character constants, `sizeof` expressions (not having a VLA operand), and floating constants that are immediate

operands of casts. When an integer constant expression is required by a preprocessor directive (such as `#if` and `#elif`), further restrictions are applied. Specifically, enumeration constants and casts are not permitted. `sizeof` expressions are permitted but are not required to be supported here.

**integer promotions** *See* conversion, integer type.

**integer suffix** *See* constant, integer.

**integer type** In C89, the following types: `char`, `short int`, `int`, and `long int`, and their unsigned counterparts. C99 added `long long int`. These types are listed in nondecreasing order of their precision. As such, one or more of the types could map to the same representation. Both signed and unsigned versions are available. The header `limits.h` can be used to determine the attributes of an implementation's integer types. All integer types are arithmetic types. *See also* `char`, plain.

**integer type conversion** *See* conversion, usual arithmetic.

**integer type, exact-width**<sup>C99</sup> *See* `intN_t`; `uintN_t`

**integer types, extended**<sup>C99</sup> Integer types that are supported by an implementation, beyond the requirements of the standard.

**integer type, minimum-width**<sup>C99</sup> *See* `int_leastN_t`; `uint_leastN_t`.

**integer type, minimum-width, fastest**<sup>C99</sup> *See* `int_fastN_t`; `uint_fastN_t`.

**integral** Prior to C99, when used as an adjective, a term that was equivalent to integer. C99 used integer instead.

**internal name** A name not exported outside of a source module. That is, it is not seen, nor resolved, by the linker. Its length of significance is at least 31 characters in C89, 63 in C99. Internal names are case-distinct. *See also* external name.

**internationalization** The process of providing support for, and integrating better with, translation or execution environments other than the original "U.S. English" mode. Commonly abbreviated as I18N. *See also* localization; locale.

**interrupt** *See* signal.

**INT\_FASTN\_MAX**<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the maximum value of the corresponding fastest minimum-width signed integer type, `int_fastN_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

`INT_FASTN_MIN`<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the minimum value of the corresponding fastest minimum-width signed integer type, `int_fastN_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

`int_fast8_t`<sup>C99</sup> See `int_fastN_t`.

`int_fast16_t`<sup>C99</sup> See `int_fastN_t`.

`int_fast32_t`<sup>C99</sup> See `int_fastN_t`.

`int_fast64_t`<sup>C99</sup> See `int_fastN_t`.

`int_fastN_t`<sup>C99</sup> A type, defined in `stdint.h`, that is the fastest signed integer type with a width of at least  $N$  bits, such that no signed integer type with lesser size has at least the specified width. For example, `int_fast16_t` denotes a signed integer type with a width of at least 16 bits. The following types must be defined: `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, and `int_fast64_t`. Other types of this form are optional. See also `INT_FASTN_MAX`; `INT_FASTN_MIN`; `uint_fastN_t`.

`int_frac_digits`<sup>C89</sup> An `lconv` structure member that is a nonnegative number representing the number of fractional digits (those after the decimal point) to be displayed in an internationally formatted monetary quantity. A value of `CHAR_MAX` indicates that the value is not available in the current locale. In the "C" locale this member must have the value `CHAR_MAX`.

`INT_LEASTN_MAX`<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the maximum value of the corresponding minimum-width signed integer type, `int_leastN_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

`INT_LEASTN_MIN`<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the minimum value of the corresponding minimum-width signed integer type, `int_leastN_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

`int_least8_t`<sup>C99</sup> See `int_leastN_t`.

`int_least16_t`<sup>C99</sup> See `int_leastN_t`.

`int_least32_t`<sup>C99</sup> See `int_leastN_t`.

`int_least64_t`<sup>C99</sup> See `int_leastN_t`.

`int_leastN_t`<sup>C99</sup> A type, defined in `stdint.h`, that is a signed integer type with a width of at least  $N$  bits, such that no signed integer type with lesser size has at least the specified width. For example, `int_least16_t` denotes a signed integer type with a width of at least 16 bits. The following

types must be defined: `int_least8_t`, `int_least16_t`, `int_least32_t`, and `int_least64_t`. Other types of this form are optional. *See also* `INT_LEASTN_MAX`; `INT_LEASTN_MIN`; `uint_leastN_t`.

`INT_MAX`<sup>C89</sup> A macro, defined in `limits.h`, that designates the maximum value for an object of type `int`. It must be at least 32,767 (16 bits). This macro expands to an integer constant expression suitable for use with a `#if` directive.

`INTMAX_C`<sup>C99</sup> A function-like macro, defined in `stdint.h`, that has the form `INTMAX_C(value)` and expands to an integer constant with the specified value and type `intmax_t`. *value* is a decimal, octal, or hexadecimal constant whose value does not exceed the limits for the corresponding type.

`INTMAX_MAX`<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the maximum value of the corresponding greatest-width signed integer type, `intmax_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

`INTMAX_MIN`<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the minimum value of the corresponding greatest-width signed integer type, `intmax_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

`intmax_t`<sup>C99</sup> A type, defined in `stdint.h`, that is a signed integer type capable of representing any value of any signed integer type, including extended types. Preprocessor arithmetic is done using this type. *See also* `INTMAX_MAX`; `INTMAX_MIN`; `uintmax_t`.

`INT_MIN`<sup>C89</sup> A macro, defined in `limits.h`, that designates the minimum value for an object of type `int`. It must be at least  $-32,767$  (16 bits). This macro expands to an integer constant expression suitable for use with a `#if` directive.

`int_n_cs_precedes`<sup>C99</sup> An `lconv` structure member that is a nonnegative number that is set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a negative internationally formatted monetary quantity. A value of `CHAR_MAX` indicates that the value is not available in the current locale. In the "C" locale this member must have the value `CHAR_MAX`.

`int_n_sep_by_space`<sup>C99</sup> An `lconv` structure member that is a nonnegative number that is set to 1 or 0 if the `int_curr_symbol` respectively is or is not separated by a space from the value for a negative internationally formatted monetary quantity. A value of `CHAR_MAX` indicates that the value is not available in the current locale. In the "C" locale this member must have the value `CHAR_MAX`.

`int_n_sign_posn`<sup>C99</sup> An `lconv` structure member that is a nonnegative number that is set to a value indicating the positioning of the `negative_sign` for a negative internationally formatted monetary quantity. The value is interpreted according to the following:

0 Parentheses surround the quantity and `int_curr_symbol`.

1 The sign string precedes the quantity and `int_curr_symbol`.

2 The sign string succeeds the quantity and `int_curr_symbol`.

3 The sign string immediately precedes the `int_curr_symbol`.

4 The sign string immediately succeeds the `int_curr_symbol`.

`CHAR_MAX` indicates that the value is not available in the current locale.

In the "C" locale this member must have the value `CHAR_MAX`.

`INTN_C`<sup>C99</sup> A function-like macro, defined in `stdint.h`, that has the form

```
INTN_C(value)
```

and expands to a signed integer constant with the specified value and type `int_leastN_t`. *value* is a decimal, octal, or hexadecimal constant whose value does not exceed the limits for the corresponding type..

`INTN_MAX`<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the maximum value of the corresponding exact-width signed integer type, `intN_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

`INTN_MIN`<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the minimum value of the corresponding exact-width signed integer type, `intN_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

`intN_t`<sup>C99</sup> A type, defined in `stdint.h`, that designates a signed integer type having width *N*, no padding bits, and a two's complement representation. For example, `int16_t` denotes a signed integer type with a width of exactly 16 bits.

Such types are optional, but if an implementation provides integer types with widths of 8, 16, 32, or 64 bits, it must define the corresponding typedef names. *See also* `INTN_MAX`; `INTN_MIN`; `uintN_t`.

`int_p_cs_precedes`<sup>C99</sup> An `lconv` structure member that is a nonnegative number that is set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a nonnegative internationally formatted monetary quantity. A value of `CHAR_MAX` indicates that the value is not available in the current locale. In the "C" locale this member must have the value `CHAR_MAX`.

**int\_p\_sep\_by\_space**<sup>C99</sup> An `lconv` structure member that is a nonnegative number that is set to 1 or 0 if the `int_curr_symbol` respectively is or is not separated by a space from the value for a nonnegative internationally formatted monetary quantity. A value of `CHAR_MAX` indicates that the value is not available in the current locale. In the "C" locale this member must have the value `CHAR_MAX`.

**int\_p\_sign\_posn**<sup>C99</sup> An `lconv` structure member that is a nonnegative number that is set to a value indicating the positioning of the `positive_sign` for a nonnegative internationally formatted monetary quantity. The value is interpreted according to the following:

- 0 Parentheses surround the quantity and `int_curr_symbol`.
- 1 The sign string precedes the quantity and `int_curr_symbol`.
- 2 The sign string succeeds the quantity and `int_curr_symbol`.
- 3 The sign string immediately precedes the `int_curr_symbol`.
- 4 The sign string immediately succeeds the `int_curr_symbol`.

`CHAR_MAX` indicates that the value is not available in the current locale.

In the "C" locale this member must have the value `CHAR_MAX`.

**INTPTR\_MAX**<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the maximum value of the corresponding pointer-holding signed integer type, `intptr_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

**INTPTR\_MIN**<sup>C99</sup> A macro, defined in `stdint.h`, that indicates the minimum value of the corresponding pointer-holding signed integer type, `intptr_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

**intptr\_t**<sup>C99</sup> An optional type, defined in `stdint.h`, that designates a signed integer type such that any valid pointer to `void` can be converted to this type and back again, with the result comparing equal to the original pointer. *See also* `INTPTR_MAX`; `INTPTR_MIN`; `uintptr_t`.

**inttypes.h**<sup>C99</sup> A header that includes `stdint.h` and, for hosted implementations, extends it with functions relating to greatest-width integers.

This header contains definitions or declarations for the following identifiers:

<i>Name</i>	<i>Purpose</i>
imaxabs	Absolute value
imaxdiv	Quotient and remainder
imaxdiv_t	Structure type used by <code>imaxdiv</code>
PRIdFAST <i>N</i>	Output conversion specifier for <code>int_fast<i>N</i>_t</code>
PRIdLEAST <i>N</i>	Output conversion specifier for <code>int_least<i>N</i>_t</code>
PRIdMAX	Output conversion specifier for <code>intmax_t</code>
PRId <i>N</i>	Output conversion specifier for <code>int<i>N</i>_t</code>
PRIdPTR	Output conversion specifier for <code>intptr_t</code>
PRIfFAST <i>N</i>	Output conversion specifier for <code>int_fast<i>N</i>_t</code>
PRIfLEAST <i>N</i>	Output conversion specifier for <code>int_least<i>N</i>_t</code>
PRIfMAX	Output conversion specifier for <code>intmax_t</code>
PRIf <i>N</i>	Output conversion specifier for <code>int<i>N</i>_t</code>
PRIfPTR	Output conversion specifier for <code>intptr_t</code>
PRIoFAST <i>N</i>	Output conversion specifier for <code>uint_fast<i>N</i>_t</code>
PRIoLEAST <i>N</i>	Output conversion specifier for <code>uint_least<i>N</i>_t</code>
PRIoMAX	Output conversion specifier for <code>uintmax_t</code>
PRIo <i>N</i>	Output conversion specifier for <code>uint<i>N</i>_t</code>
PRIoPTR	Output conversion specifier for <code>uintptr_t</code>
PRIfuFAST <i>N</i>	Output conversion specifier for <code>uint_fast<i>N</i>_t</code>
PRIfuLEAST <i>N</i>	Output conversion specifier for <code>uint_least<i>N</i>_t</code>
PRIfuMAX	Output conversion specifier for <code>uintmax_t</code>
PRIfu <i>N</i>	Output conversion specifier for <code>uint<i>N</i>_t</code>
PRIfuPTR	Output conversion specifier for <code>uintptr_t</code>
PRIfxFAST <i>N</i>	Output conversion specifier for <code>uint_fast<i>N</i>_t</code>
PRIfxFAST <i>N</i>	Output conversion specifier for <code>uint_fast<i>N</i>_t</code>
PRIfxLEAST <i>N</i>	Output conversion specifier for <code>uint_least<i>N</i>_t</code>
PRIfxLEAST <i>N</i>	Output conversion specifier for <code>uint_least<i>N</i>_t</code>
PRIfxMAX	Output conversion specifier for <code>uintmax_t</code>
PRIfxMAX	Output conversion specifier for <code>uintmax_t</code>
PRIfx <i>N</i>	Output conversion specifier for <code>uint<i>N</i>_t</code>
PRIfx <i>N</i>	Output conversion specifier for <code>uint<i>N</i>_t</code>
PRIfxPTR	Output conversion specifier for <code>uintptr_t</code>
PRIfxPTR	Output conversion specifier for <code>uintptr_t</code>
SCNdFAST <i>N</i>	Input conversion specifier for <code>int_fast<i>N</i>_t</code>
SCNdLEAST <i>N</i>	Input conversion specifier for <code>int_least<i>N</i>_t</code>
SCNdMAX	Input conversion specifier for <code>intmax_t</code>
SCNd <i>N</i>	Input conversion specifier for <code>int<i>N</i>_t</code>
SCNdPTR	Input conversion specifier for <code>intptr_t</code>
SCNiFAST <i>N</i>	Input conversion specifier for <code>int_fast<i>N</i>_t</code>
SCNiLEAST <i>N</i>	Input conversion specifier for <code>int_least<i>N</i>_t</code>
SCNiMAX	Input conversion specifier for <code>intmax_t</code>
SCNi <i>N</i>	Input conversion specifier for <code>int<i>N</i>_t</code>
SCNiPTR	Input conversion specifier for <code>intptr_t</code>

<i>Name</i>	<i>Purpose</i>
SCNoFAST <i>N</i>	Input conversion specifier for <code>uint_fast<i>N</i>_t</code>
SCNoLEAST <i>N</i>	Input conversion specifier for <code>uint_least<i>N</i>_t</code>
SCNoMAX	Input conversion specifier for <code>uintmax_t</code>
SCNo <i>N</i>	Input conversion specifier for <code>uint<i>N</i>_t</code>
SCNoPTR	Input conversion specifier for <code>uintptr_t</code>
SCNuFAST <i>N</i>	Input conversion specifier for <code>uint_fast<i>N</i>_t</code>
SCNuLEAST <i>N</i>	Input conversion specifier for <code>uint_least<i>N</i>_t</code>
SCNuMAX	Input conversion specifier for <code>uintmax_t</code>
SCNu <i>N</i>	Input conversion specifier for <code>uint<i>N</i>_t</code>
SCNuPTR	Input conversion specifier for <code>uintptr_t</code>
SCNxFAST <i>N</i>	Input conversion specifier for <code>uint_fast<i>N</i>_t</code>
SCNxLEAST <i>N</i>	Input conversion specifier for <code>uint_least<i>N</i>_t</code>
SCNxMAX	Input conversion specifier for <code>uintmax_t</code>
SCNx <i>N</i>	Input conversion specifier for <code>uint<i>N</i>_t</code>
SCNxPTR	Input conversion specifier for <code>uintptr_t</code>
<code>strtoimax</code>	Convert string to signed integer
<code>strtoumax</code>	Convert string to unsigned integer
<code>wcstoimax</code>	Convert wide string to signed integer
<code>wcstoumax</code>	Convert wide string to unsigned integer

In these names, *N* represents the width of the type in bits.

*See* future library directions.

**\_IOFBF** A macro, defined in `stdio.h`, that can be used as the third argument to `setvbuf` to indicate that a stream is fully buffered. *See also* `_IOLBF`; `_IONBF`.

**\_IOLBF** A macro, defined in `stdio.h`, that can be used as the third argument to `setvbuf` to indicate that a stream is line buffered. *See also* `_IOFBF`; `_IONBF`.

**\_IONBF** A macro, defined in `stdio.h`, that can be used as the third argument to `setvbuf` to indicate that a stream is not buffered. *See also* `_IOFBF`; `_IOLBF`.

**is prefix** *See* future library directions.

**isalnum** A locale-specific function that tests if its argument *c* is alphabetic (`isalpha`) or decimal numeric (`isdigit`).

```
#include <ctype.h>
int isalnum(int c);
```

A zero value is returned for false, a nonzero for true. *See also* `iswalnum`.

**isalpha** A locale-specific function that tests if its argument `c` is an alphabetic character.

```
#include <ctype.h>
int isalpha(int c);
```

In the "C" locale, this means that either `isupper` or `islower` is true. In other locales, other implementation-defined characters may be included in the `isalpha` set, provided they are not in the `iscntrl`, `isdigit`, `ispunct`, or `isspace` sets as well. *See also* `iswalpha`.

A zero value is returned for false, a nonzero for true.

**isblank**<sup>C99</sup> A function that tests for any character that is a standard blank character or is one of a locale-specific set of characters for which `isspace` is true and which is used to separate words within a line of text.

```
#include <ctype.h>
int isblank(int c);
```

The standard blank characters are the following: space (' ') and horizontal tab ('\t'). In the "C" locale, `isblank` returns true for the standard blank characters only. *See also* `isspace`; `iswblank`; `iswspace`.

**iscntrl** A function that tests to see if its argument `c` is one of an implementation-defined set of control characters.

```
#include <ctype.h>
int iscntrl(int c);
```

In ASCII, the control characters have values 0 through 0x1f, and 0x7f. In EBCDIC, the control characters have values less than 0x40 (except that 0x29, 0x30, 0x31, and 0x3e are not control characters.)

A zero value is returned for false, a nonzero for true. *See also* `iswcntrl`.

**isdigit** A function that tests to see if its argument `c` is one of the decimal digits 0–9.

```
#include <ctype.h>
int isdigit(int c);
```

A zero value is returned for false, a nonzero for true. *See also* `iswdigit`.

**isfinite**<sup>C99</sup> A macro, defined in `math.h`, that indicates whether its argument has a finite (that is, not infinite or NaN) value.

```
#include <math.h>
int isfinite(real-floating x);
```

A zero value is returned for false, a nonzero for true.

**isgraph** A locale-specific function that tests if its argument *c* is any printable character except a space.

```
#include <ctype.h>
int isgraph(int c);
```

A zero value is returned for false, a nonzero for true. *See also iswgraph.*

**isgreater**<sup>C99</sup> A macro, defined in *math.h*, that indicates whether its first argument is greater than its second.

```
#include <math.h>
int isgreater(real-floating x, real-floating y);
```

A zero value is returned for false, a nonzero for true. Unlike the corresponding relational operator *>*, this function does not raise the “invalid” floating-point exception when *x* and *y* are unordered (i.e., either is a NaN).

**isgreaterequal**<sup>C99</sup> A macro, defined in *math.h*, that indicates whether its first argument is greater than or equal to its second.

```
#include <math.h>
int isgreaterequal(real-floating x, real-floating y);
```

A zero value is returned for false, a nonzero for true. Unlike the corresponding relational operator *>=*, this function does not raise the “invalid” floating-point exception when *x* and *y* are unordered (i.e., either is a NaN).

**isinf**<sup>C99</sup> A macro, defined in *math.h*, that indicates whether its argument is a positive or negative infinity.

```
#include <math.h>
int isinf(real-floating x);
```

A zero value is returned for false, a nonzero for true.

**isless**<sup>C99</sup> A macro, defined in *math.h*, that indicates whether its first argument is less than its second.

```
#include <math.h>
int isless(real-floating x, real-floating y);
```

A zero value is returned for false, a nonzero for true. Unlike the corresponding relational operator `<`, this function does not raise the “invalid” floating-point exception when `x` and `y` are unordered (i.e., either is a NaN).

**islessequal**<sup>C99</sup> A macro, defined in `math.h`, that indicates whether its first argument is less than or equal to its second.

```
#include <math.h>
int islessequal(real-floating x, real-floating y);
```

A zero value is returned for false, a nonzero for true. Unlike the corresponding relational operator `<=`, this function does not raise the “invalid” floating-point exception when `x` and `y` are unordered (i.e., either is a NaN).

**islessgreater**<sup>C99</sup> A macro, defined in `math.h`, that indicates whether its first argument is either less than or greater than its second.

```
#include <math.h>
int islessgreater(real-floating x, real-floating y);
```

A zero value is returned for false, a nonzero for true. It behaves like `x < y || x > y`, but it does not raise the “invalid” floating-point exception when `x` and `y` are unordered (i.e., either is a NaN).

**islower** A locale-specific function that tests if its argument `c` is a lowercase alphabetic character.

```
#include <ctype.h>
int islower(int c);
```

In the “C” locale, lowercase letters are the Roman letters `a–z` inclusive. In other locales, other implementation-defined characters may be included in the `isalpha` set, provided they are not in the `iscntrl`, `isdigit`, `ispunct`, or `isspace` sets as well.

A zero value is returned for false, a nonzero for true. *See also* `iswlower`.

**isnan**<sup>C99</sup> A macro, defined in `math.h`, that indicates whether its argument is a NaN.

```
#include <math.h>
int isnan(real-floating x);
```

A zero value is returned for false, a nonzero for true.

**isnormal**<sup>C99</sup> A macro, defined in `math.h`, that indicates whether its argument is normal (that is, not zero, subnormal, infinite, or NaN).

```
#include <math.h>
int isnormal(real-floating x);
```

A zero value is returned for false, a nonzero for true.

**ISO** Acronym for the International Organization for Standardization, the international counterpart of national standards bodies. A Joint Technical Committee, JTC 1, has been set up between ISO and IEC (International Electrotechnical Commission) to produce standards for programming languages, among other things. JTC 1 standards are most often referred to as ISO Standards. *See also* ISO C.

**ISO C** The formal definition of the C language, preprocessor, and run-time library as accepted by ISO in 1989 and then again in 1999. This standard was produced by committee WG14. *See also* Standard C.

**ISO 4217 Codes for the representation for currencies and funds** A standard referred to when the currency part of the locale structure type `lconv` was defined. *See also* `int_curr_symbol`.

**ISO 646 Invariant Code Set** A coding scheme used primarily in Europe that is missing nine critical source characters needed in C programming. *See also* digraph; trigraph.

**iso646.h**<sup>C95</sup> A header that provides a family of macros that allow programmers using source character sets (such as ISO 646) missing certain characters necessary for writing C programs, to enter those characters using identifiers instead. The macros defined here are `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor`, and `xor_eq`. Note that in C++, these are implemented as keywords instead.

**ISO/IEC 10646** A character set family. Most people use it to mean ISO/IEC 10646.UCS-2.

**ISO/IEC 10646.UCS-2** A two-byte per character character set based on Unicode.

**ISO/IEC 10646.UCS-4** A four-byte per character character set.

**ISO/IEC 646** *See* digraph; ISO 646 Invariant Code Set; trigraph.

**isprint** A locale-specific function that tests if its argument `c` is any printable character, including a space.

```
#include <ctype.h>
int isprint(int c);
```

A zero value is returned for false, a nonzero for true. *See also* `iswprint`. `isprint` subsumes `isgraph`'s functionality.

**ispunct** A locale-specific function that tests if its argument `c` is any printable character, except a space, or any character for which `isalnum` tests true.

```
#include <ctype.h>
int ispunct(int c);
```

A zero value is returned for false, a nonzero for true. *See also* `iswpunct`.

**isspace** A locale-specific function that tests if its argument `c` is any of the white space characters.

```
#include <ctype.h>
int isspace(int c);
```

In the "C" locale, the set of characters is space, form feed, new-line, carriage return, horizontal tab, and vertical tab. In other locales, other implementation-defined characters may be added to the above set, provided they do not test true for `isalnum`.

A zero value is returned for false, a nonzero for true. *See also* `isblank`; `iswblank`; `iswspace`.

**isunordered**<sup>C99</sup> A macro, defined in `math.h`, that indicates whether its arguments are unordered (i.e., either is a NaN).

```
#include <math.h>
int isunordered(real-floating x, real-floating y);
```

A zero value is returned for false, a nonzero for true.

**isupper** A locale-specific function that tests if its argument `c` is an uppercase alphabetic character.

```
#include <ctype.h>
int isupper(int c);
```

In the "C" locale, uppercase letters are the Roman letters A–Z inclusive. In other locales, other implementation-defined characters may be included in the `isalpha` set, provided they are not in the `iscntrl`, `isdigit`, `ispunct`, or `isspace` sets as well.

A zero value is returned for false, a nonzero for true. *See also* `iswupper`.

`isxdigit` A function that tests if its argument `c` is a hexadecimal digit. The set of valid characters are the digits 0–9 and the letters A–F and a–f.

```
#include <ctype.h>
int isxdigit(int c);
```

A zero value is returned for false, a nonzero for true. *See also* `iswxdigit`.

`iswalnum`<sup>C95</sup> A locale-specific function that tests if its wide character argument `wc` is alphabetic (`iswalpha`) or decimal numeric (`iswdigit`).

```
#include <wctype.h>
int iswalnum(wint_t wc);
```

A zero value is returned for false, a nonzero for true. *See also* `isalnum`.

`iswalpha`<sup>C95</sup> A locale-specific function that tests if its wide character argument `wc` is an alphabetic character.

```
#include <wctype.h>
int iswalpha(wint_t wc);
```

In the "C" locale, this means that either `iswupper` or `iswlower` is true. In other locales, other implementation-defined characters may be included in the `isalpha` set, provided they are not in the `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` sets as well. *See also* `isalpha`.

A zero value is returned for false, a nonzero for true.

`iswblank`<sup>C99</sup> A function that tests if its wide character argument `wc` is a blank wide character or is one of a locale-specific set of wide characters for which `iswspace` is true and that is used to separate words within a line of text.

```
#include <wctype.h>
int iswblank(wint_t wc);
```

The standard blank wide characters are: space (L' ') and horizontal tab (L'\t'). In the "C" locale, `iswblank` returns true only for the standard blank characters. *See also* `iswspace`.

`iswcntrl`<sup>C95</sup> A function that tests to see if its wide character argument `wc` is one of an implementation-defined set of control characters.

```
#include <wctype.h>
int iswcntrl(wint_t wc);
```

A zero value is returned for false, a nonzero for true. *See also* `iscntrl`.

`iswctype`<sup>C95</sup> A function that tests whether the wide character `wc` has the property described by `desc`.

```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t desc);
```

The setting of the `LC_CTYPE` category when this function is called must be the same as that during the call to `wctype` that produced the value `desc`.

A zero value is returned for false, a nonzero for true.

`iswdigit`<sup>C95</sup> A function that tests to see if its wide character argument `wc` is one of the decimal digits 0–9.

```
#include <wctype.h>
int iswdigit(wint_t wc);
```

A zero value is returned for false, a nonzero for true. *See also* `isdigit`.

`iswgraph`<sup>C95</sup> A locale-specific function that tests if its wide character argument `wc` is any printable character except a space.

```
#include <wctype.h>
int iswgraph(wint_t wc);
```

A zero value is returned for false, a nonzero for true. *See also* `isgraph`.

`iswlower`<sup>C95</sup> A locale-specific function that tests if its wide character argument `wc` is a lowercase alphabetic character.

```
#include <wctype.h>
int iswlower(wint_t wc);
```

In the "C" locale, lowercase letters are the Roman letters a–z inclusive. In other locales, other implementation-defined characters may be included in the `iswalphabet` set, provided they are not in the `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` sets as well.

A zero value is returned for false, a nonzero for true. *See also* `islower`.

**iswprint**<sup>C95</sup> A locale-specific function that tests if its wide character argument `wc` is any printable character, including a space.

```
#include <wctype.h>
int iswprint(wint_t wc);
```

A zero value is returned for false, a nonzero for true. *See also* `isprint`. `isprint` subsumes `isgraph`'s functionality.

**iswpunct**<sup>C95</sup> A locale-specific function that tests if its wide character argument `wc` is any printable character except a space, or any character for which neither `iswspace` nor `iswalnum` tests true.

```
#include <wctype.h>
int iswpunct(wint_t wc);
```

A zero value is returned for false, a nonzero for true. *See also* `ispunct`.

**iswspace**<sup>C95</sup> A locale-specific function that tests if its wide character argument `wc` is any of the white space characters.

```
#include <ctype.h>
int isspace(int c);
```

In the "C" locale, the set of characters is space, form feed, new-line, carriage return, horizontal tab, and vertical tab. In other locales, other implementation-defined characters may be added to the above set, provided they do not test true for `iswalnum`, `iswgraph`, or `iswpunct`.

A zero value is returned for false, a nonzero for true. *See also* `isblank`, `isspace`, `iswblank`.

**iswupper**<sup>C95</sup> A locale-specific function that tests if its wide character argument `wc` is an uppercase alphabetic character.

```
#include <wctype.h>
int iswupper(wint_t wc);
```

In the "C" locale, uppercase letters are the Roman letters A–Z inclusive. In other locales, other implementation-defined characters may be included in the `iswalpha` set, provided they are not in the `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` sets as well.

A zero value is returned for false, a nonzero for true. *See also* `isupper`.

`iswxdigit`<sup>C95</sup> A function that tests if its wide character argument `wc` is a hexadecimal digit. The set of valid characters are the digits 0–9 and the letters A–F and a–f.

```
#include <wctype.h>
int iswxdigit(wint_t wc);
```

A zero value is returned for false, a nonzero for true. *See also* `isxdigit`.

**iteration statements** The `while`, `do`, and `for` statements.

◇ ◇ ◇