GCC __builtin_

GCC provides a large number of built-in functions other than the ones mentioned above. Some of these are for internal use in the processing of exceptions or variable-length argument lists and will not be documented here because they may change from time to time; we do not recommend general use of these functions.

The remaining functions are provided for optimization purposes.

GCC includes built-in versions of many of the functions in the standard C library. The versions prefixed with `__builtin_` will always be treated as having the same meaning as the C library function even if you specify the `-fno-builtin` option. (see C Dialect Options) Many of these functions are only optimized in certain cases; if they are not optimized in a particular case, a call to the library function will be emitted.

Outside strict ISO C mode (`-ansi`, `-std=c89` or `-std=c99`), the functions `_exit`, `alloca`, `bcmp`, `bzero`, `dcgettext`, `dgettext`, `dremf`, `dreml`, `drem`, `exp10f`, `exp10l`, `exp10`, `ffsll`, `ffsl`, `ffs`, `fprintf_unlocked`, `fputs_unlocked`, `gammaf`, `gammal`, `gamma`, `gettext`, `index`, `isascii`, `j0f`, `j0l`, `j0`, `j1f`, `j1l`, `j1`, `jnf`, `jnl`, `jn`, `mempcpy`, `pow10f`, `pow10l`, `pow10`, `printf_unlocked`, `rindex`, `scalbf`, `scalbl`, `scalb`, `signbit`, `signbitf`, `signbitl`, `significandf`, `significandl`, `significand`, `sincosf`, `sincosl`, `sincos`, `stpcpy`, `strdup`, `strfmon`, `toascii`, `y0f`, `y0l`, `y0`, `y1f`, `y1l`, `y1`, `ynf`, `ynl` and `yn` may be handled as built-in functions. All these functions have corresponding versions prefixed with `__builtin_`, which may be used even in strict C89 mode.

The ISO C99 functions `_Exit`, `acoshf`, `acoshl`, `acosh`, `asinhf`, `asinhl`, `asinh`, `atanhf`, `atanhl`, `atanh`, `cabsf`, `cabsl`, `cabs`, `cacosf`, `cacoshf`, `cacoshl`, `cacosh`, `cacosl`, `cacos`, `cargf`, `cargl`, `carg`, `casinf`, `casinhf`, `casinhl`, `casinh`, `casinl`, `casin`, `catanf`, `catanhf`, `catanhl`, `catanh`, `catanl`, `catan`, `cbrtf`, `cbrtl`, `cbrt`, `ccosf`, `ccoshf`, `ccoshl`, `ccosh`, `ccosl`, `ccos`, `cexpf`, `cexpl`, `cexp`, `cimagf`, `cimagl`, `cimag`, `conjf`, `conjl`, `conj`, `copysignf`, `copysignl`, `copysign`, `cpowf`, `cpowl`, `cpow`, `cprojf`, `cprojl`, `cproj`, `crealf`, `creall`, `creal`, `csinf`, `csinhf`, `csinhl`, `csinh`, `csinl`, `csin`, `csqrtf`, `csqrtl`, `csqrt`, `ctanf`, `ctanhf`, `ctanhl`, `ctanh`, `ctanl`, `ctan`, `erfcf`, `erfcl`, `erfc`, `erff`, `erfl`, `erf`, `exp2f`, `exp2l`, `exp2`, `expm1f`, `expm1l`, `expm1`, `fdimf`, `fdiml`, `fdim`, `fmaf`, `fmal`, `fmaxf`, `fmaxl`, `fmax`, `fma`, `fminf`, `fminl`, `fmin`, `hypotf`, `hypotl`, `hypot`, `ilogbf`, `ilogbl`, `ilogb`, `imaxabs`, `isblank`, `iswblank`, `lgammaf`, `lgammal`, `lgamma`, `llabs`, `llrintf`, `llrintl`, `llrint`, `llroundf`, `llroundl`, `llround`, `log1pf`, `log1pl`, `log1p`, `log2f`, `log2l`, `log2`, `logbf`, `logbl`, `logb`, `lrintf`, `lrintl`, `lrint`, `lroundf`, `lroundl`, `lround`, `nearbyintf`, `nearbyintl`, `nearbyint`, `nextafterf`, `nextafterl`, `nextafter`, `nexttowardf`, `nexttowardl`, `nexttoward`, `remainderf`, `remainderl`, `remainder`, `remquof`, `remquol`, `remquo`, `rintf`, `rintl`, `rint`, `roundf`, `roundl`, `round`, `scalblnf`, `scalblnl`, `scalbln`, `scalbnf`, `scalbnl`, `scalbn`, `snprintf`, `tgammaf`, `tgammal`, `tgamma`, `truncf`, `truncl`, `trunc`, `vfscanf`, `vscanf`, `vsnprintf` and `vsscanf` are handled as built-in functions except in strict ISO C90 mode (`-ansi` or `-std=c89`).

There are also built-in versions of the ISO C99 functions `acosf`, `acosl`, `asinf`, `asinl`, `atan2f`, `atan2l`, `atanf`, `atanl`, `ceilf`, `ceill`, `cosf`, `coshf`, `coshl`, `cosl`, `expf`, `expl`, `fabsf`, `fabsl`, `floorf`, `floorl`, `fmodf`, `fmodl`, `frexpf`, `frexpl`, `ldexpf`, `ldexpl`, `log10f`, `log10l`, `logf`, `logl`, `modfl`, `modf`, `powf`, `powl`, `sinf`, `sinhf`, `sinhl`, `sinl`, `sqrtf`, `sqrtl`, `tanf`, `tanhf`, `tanhl` and `tanl` that are recognized in any mode since ISO C90 reserves these names for the purpose to which ISO C99 puts them. All these functions have corresponding versions prefixed with `__builtin_`.

The ISO C94 functions `iswalnum`, `iswalpha`, `iswcntrl`, `iswdigit`, `iswgraph`, `iswlower`, `iswprint`, `iswpunct`, `iswspace`, `iswupper`, `iswxdigit`, `towlower` and `towupper` are handled as built-in functions except in strict ISO C90 mode (`-ansi` or `-std=c89`).

The ISO C90 functions `abort`, `abs`, `acos`, `asin`, `atan2`, `atan`, `calloc`, `ceil`, `cosh`, `cos`, `exit`, `exp`, `fabs`, `floor`, `fmod`, `fprintf`, `fputs`, `frexp`, `fscanf`, `isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`, `tolower`, `toupper`, `labs`, `ldexp`, `log10`, `log`, `malloc`, `memcmp`, `memcpy`, `memset`, `modf`, `pow`, `printf`, `putchar`, `puts`, `scanf`, `sinh`, `sin`, `snprintf`, `sprintf`, `sqrt`, `sscanf`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strcspn`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `tanh`, `tan`, `vfprintf`, `vprintf` and `vsprintf` are all recognized as built-in functions unless `-fno-builtin` is specified (or `-fno-builtin-`*`function`* is specified for an individual function). All of these functions have corresponding versions prefixed with `__builtin_`.

GCC provides built-in versions of the ISO C99 floating point comparison macros that avoid raising exceptions for unordered operands. They have the same names as the standard macros (`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, and `isunordered`) , with `__builtin_` prefixed. We intend for a library implementor to be able to simply `#define` each standard macro to its built-in equivalent.

— Built-in Function: int **__builtin_types_compatible_p** (*type1*, *type2*)

> You can use the built-in function `__builtin_types_compatible_p` to determine whether two types are the same.
>
> This built-in function returns 1 if the unqualified versions of the types *type1* and *type2* (which are types, not expressions) are compatible, 0 otherwise. The result of this built-in function can be used in integer constant expressions.
>
> This built-in function ignores top level qualifiers (e.g., `const`, `volatile`). For example, `int` is equivalent to `const int`.

The type `int[]` and `int[5]` are compatible. On the other hand, `int` and `char *` are not compatible, even if the size of their types, on the particular architecture are the same. Also, the amount of pointer indirection is taken into account when determining similarity. Consequently, `short *` is not similar to `short **`. Furthermore, two types that are typedefed are considered compatible if their underlying types are compatible.

An `enum` type is not considered to be compatible with another `enum` type even if both are compatible with the same integer type; this is what the C standard specifies. For example, `enum {foo, bar}` is not similar to `enum {hot, dog}`.

You would typically use this function in code whose execution varies depending on the arguments' types. For example:

```
#define foo(x)                                                  \
 ({                                                           \
  typeof (x) tmp;                                             \
  if (__builtin_types_compatible_p (typeof (x), long double)) \
   tmp = foo_long_double (tmp);                              \
  else if (__builtin_types_compatible_p (typeof (x), double)) \
   tmp = foo_double (tmp);                                   \
  else if (__builtin_types_compatible_p (typeof (x), float))  \
   tmp = foo_float (tmp);                                    \
  else                                                        \
   abort ();                                                 \
  tmp;                                                        \
 })
```

Note: This construct is only available for C.

— Built-in Function: *type* **__builtin_choose_expr** (*const_exp*, *exp1*, *exp2*)

You can use the built-in function `__builtin_choose_expr` to evaluate code depending on the value of a constant expression. This built-in function returns *exp1* if *const_exp*, which is a constant expression that must be able to be determined at compile time, is nonzero. Otherwise it returns 0.

This built-in function is analogous to the `` `? :' `` operator in C, except that the expression returned has its type unaltered by promotion rules. Also, the built-in function does not evaluate the expression that was not chosen. For example, if *const_exp* evaluates to true, *exp2* is not evaluated even if it has side-effects.

This built-in function can return an lvalue if the chosen argument is an lvalue.

If *exp1* is returned, the return type is the same as *exp1*'s type. Similarly, if *exp2* is returned, its return type is the same as *exp2*.

Example:

```
    #define
foo(x)                                                    \
    __builtin_choose_expr (                                      \
     __builtin_types_compatible_p (typeof (x), double),          \
     foo_double (x),                                             \
     __builtin_choose_expr (                                     \
      __builtin_types_compatible_p (typeof (x), float),          \
      foo_float (x),                                             \
      /* The void expression results in a compile-time error  \
        when assigning the result to something.  */          \
      (void)0))
```

Note: This construct is only available for C. Furthermore, the unused expression (*exp1* or *exp2* depending on the value of *const_exp*) may still generate syntax errors. This may change in future revisions.

— Built-in Function: int **__builtin_constant_p** (*exp*)

You can use the built-in function `__builtin_constant_p` to determine if a value is known to be constant at compile-time and hence that GCC can perform constant-folding on expressions involving that value. The argument of the function is the value to test. The function returns the integer 1 if the argument is known to be a compile-time constant and 0 if it is not known to be a compile-time constant. A return of 0 does not indicate that the value is *not* a constant, but merely that GCC cannot prove it is a constant with the specified value of the `-O` option.

You would typically use this function in an embedded application where memory was a critical resource. If you have some complex calculation, you may want it to be folded if it involves constants, but need to call a function if it does not. For example:

```
#define Scale_Value(X)      \
 (__builtin_constant_p (X) \
 ? ((X) * SCALE + OFFSET) : Scale (X))
```

You may use this built-in function in either a macro or an inline function. However, if you use it in an inlined function and pass an argument of the function as the argument to the built-in, GCC will never return 1 when you call the inline function with a string constant or compound literal (see Compound Literals) and will not return 1 when you pass a constant numeric value to the inline function unless you specify the `-O` option.

You may also use `__builtin_constant_p` in initializers for static data. For instance, you can write

```
static const int table[] = {
  __builtin_constant_p (EXPRESSION) ? (EXPRESSION) : -1,
  /* ... */
};
```

This is an acceptable initializer even if *EXPRESSION* is not a constant expression. GCC must be more conservative about evaluating the built-in in this case, because it has no opportunity to perform optimization.

Previous versions of GCC did not accept this built-in in data initializers. The earliest version where it is completely safe is 3.0.1.

— Built-in Function: long **__builtin_expect** (*long exp, long c*)

You may use `__builtin_expect` to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this (`-fprofile-arcs`), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

The return value is the value of *exp*, which should be an integral expression. The value of *c* must be a compile-time constant. The semantics of the built-in are that it is expected that *exp* == *c*. For example:

```
if (__builtin_expect (x, 0))
 foo ();
```

would indicate that we do not expect to call `foo`, since we expect `x` to be zero. Since you are limited to integral expressions for *exp*, you should use constructions such as

```
if (__builtin_expect (ptr != NULL, 1))
 error ();
```

when testing pointer or floating-point values.

— Built-in Function: void **__builtin_prefetch** (*const void *addr, ...*)

This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to `__builtin_prefetch` into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions will be generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of *addr* is the address of the memory to prefetch. There are two optional arguments, *rw* and *locality*. The value of *rw* is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value *locality* must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

```
for (i = 0; i < n; i++)
 {
  a[i] = a[i] + b[i];
  __builtin_prefetch (&a[i+j], 1, 1);
  __builtin_prefetch (&b[i+j], 0, 1);
  /* ... */
```

```
          }
```

Data prefetch does not generate faults if *addr* is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` will not fault if `p->next` is not a valid address, but evaluation will fault if `p` is not a valid address.

If the target does not support data prefetch, the address expression is evaluated if it includes side effects but no other code is generated and GCC does not issue a warning.

— Built-in Function: double **__builtin_huge_val** (*void*)

Returns a positive infinity, if supported by the floating-point format, else `DBL_MAX`. This function is suitable for implementing the ISO C macro `HUGE_VAL`.

— Built-in Function: float **__builtin_huge_valf** (*void*)

Similar to `__builtin_huge_val`, except the return type is `float`.

— Built-in Function: long double **__builtin_huge_vall** (*void*)

Similar to `__builtin_huge_val`, except the return type is `long double`.

— Built-in Function: double **__builtin_inf** (*void*)

Similar to `__builtin_huge_val`, except a warning is generated if the target floating-point format does not support infinities.

— Built-in Function: float **__builtin_inff** (*void*)

Similar to `__builtin_inf`, except the return type is `float`. This function is suitable for implementing the ISO C99 macro `INFINITY`.

— Built-in Function: long double **__builtin_infl** (*void*)

Similar to `__builtin_inf`, except the return type is `long double`.

— Built-in Function: double **__builtin_nan** (*const char *str*)

This is an implementation of the ISO C99 function `nan`.

Since ISO C99 defines this function in terms of `strtod`, which we do not implement, a description of the parsing is in order. The string is parsed as by `strtol`; that is, the base is recognized by leading `` `0' `` or `` `0x' `` prefixes. The number parsed is placed in the significand such that the least significant bit of the number is at the least significant bit of the significand. The number is truncated to fit the significand field provided. The significand is forced to be a quiet NaN.

This function, if given a string literal, is evaluated early enough that it is considered a compile-time constant.

— Built-in Function: float **__builtin_nanf** (*const char *str*)

Similar to `__builtin_nan`, except the return type is `float`.

— Built-in Function: long double **__builtin_nanl** (*const char *str*)

Similar to `__builtin_nan`, except the return type is `long double`.

— Built-in Function: double **__builtin_nans** (*const char *str*)

Similar to `__builtin_nan`, except the significand is forced to be a signaling NaN. The `nans` function is proposed by WG14 N965.

— Built-in Function: float **__builtin_nansf** (*const char *str*)

Similar to `__builtin_nans`, except the return type is `float`.

— Built-in Function: long double **__builtin_nansl** (*const char *str*)

Similar to `__builtin_nans`, except the return type is `long double`.

— Built-in Function: int **__builtin_ffs** (*unsigned int x*)

Returns one plus the index of the least significant 1-bit of *x*, or if *x* is zero, returns zero.

— Built-in Function: int **__builtin_clz** (*unsigned int x*)

Returns the number of leading 0-bits in *x*, starting at the most significant bit position. If *x* is 0, the result is undefined.

— Built-in Function: int **__builtin_ctz** (*unsigned int x*)

Returns the number of trailing 0-bits in *x*, starting at the least significant bit position. If *x* is 0, the result is undefined.

— Built-in Function: int **__builtin_popcount** (*unsigned int x*)

Returns the number of 1-bits in *x*.

— Built-in Function: int **__builtin_parity** (*unsigned int x*)

Returns the parity of *x*, i.e. the number of 1-bits in *x* modulo 2.

— Built-in Function: int **__builtin_ffsl** (*unsigned long*)

Similar to `__builtin_ffs`, except the argument type is `unsigned long`.

— Built-in Function: int **__builtin_clzl** (*unsigned long*)

Similar to `__builtin_clz`, except the argument type is `unsigned long`.

— Built-in Function: int **__builtin_ctzl** (*unsigned long*)

Similar to `__builtin_ctz`, except the argument type is `unsigned long`.

— Built-in Function: int **__builtin_popcountl** (*unsigned long*)

Similar to `__builtin_popcount`, except the argument type is `unsigned long`.

— Built-in Function: int **__builtin_parityl** (*unsigned long*)

Similar to `__builtin_parity`, except the argument type is `unsigned long`.

— Built-in Function: int **__builtin_ffsll** (*unsigned long long*)

Similar to `__builtin_ffs`, except the argument type is `unsigned long long`.

— Built-in Function: int **__builtin_clzll** (*unsigned long long*)

Similar to `__builtin_clz`, except the argument type is `unsigned long long`.

— Built-in Function: int **__builtin_ctzll** (*unsigned long long*)

Similar to `__builtin_ctz`, except the argument type is `unsigned long long`.

— Built-in Function: int **__builtin_popcountll** (*unsigned long long*)

Similar to `__builtin_popcount`, except the argument type is `unsigned long long`.

— Built-in Function: int **__builtin_parityll** (*unsigned long long*)

Similar to `__builtin_parity`, except the argument type is `unsigned long long`.

— Built-in Function: double **__builtin_powi** (*double, int*)

Returns the first argument raised to the power of the second. Unlike the `pow` function no guarantees about precision and rounding are made.

— Built-in Function: float **__builtin_powif** (*float, int*)

Similar to `__builtin_powi`, except the argument and return types are `float`.

— Built-in Function: long double **__builtin_powil** (*long double, int*)

Similar to `__builtin_powi`, except the argument and return types are `long double`.