Next: Target Builtins, Previous: Cilk Plus Builtins, Up: C Extensions   [Contents][Index]

6.59 Other Built-in Functions Provided by GCC

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 are not 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.

With the exception of built-ins that have library equivalents such as the standard C library functions discussed below, or that expand to library calls, GCC built-in functions are always expanded inline and thus do not have corresponding entry points and their address cannot be obtained. Attempting to use them in an expression other than a function call results in a compile-time error.

GCC includes built-in versions of many of the functions in the standard C library. These functions come in two forms: one whose names start with the `__builtin_` prefix, and the other without. Both forms have the same type (including prototype), the same address (when their address is taken), and the same meaning as the C library functions 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 is emitted.

Outside strict ISO C mode (`-ansi`, `-std=c90`, `-std=c99` or `-std=c11`), the functions `_exit`, `alloca`, `bcmp`, `bzero`, `dcgettext`, `dgettext`, `dremf`, `dreml`, `drem`, `exp10f`, `exp10l`, `exp10`, `ffsll`, `ffsl`, `ffs`, `fprintf_unlocked`, `fputs_unlocked`, `gammaf`, `gammal`, `gamma`, `gammaf_r`, `gammal_r`, `gamma_r`, `gettext`, `index`, `isascii`, `j0f`, `j0l`, `j0`, `j1f`, `j1l`, `j1`, `jnf`, `jnl`, `jn`, `lgammaf_r`, `lgammal_r`, `lgamma_r`, `mempcpy`, `pow10f`, `pow10l`, `pow10`, `printf_unlocked`, `rindex`, `scalbf`, `scalbl`, `scalb`, `signbit`, `signbitf`, `signbitl`, `signbitd32`, `signbitd64`, `signbitd128`, `significandf`, `significandl`, `significand`, `sincosf`, `sincosl`, `sincos`, `stpcpy`, `stpncpy`, `strcasecmp`, `strdup`, `strfmon`, `strncasecmp`, `strndup`, `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 C90 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`, `clogf`, `clogl`, `clog`, `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=c90`).

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_`.

There are also built-in functions `__builtin_fabsf`*n*, `__builtin_fabsf`*n*`x`, `__builtin_copysignf`*n* and `__builtin_copysignf`*n*`x`, corresponding to the TS 18661-3 functions `fabsf`*n*, `fabsf`*n*`x`, `copysignf`*n* and `copysignf`*n*`x`, for supported types `_Float`*n* and `_Float`*n*`x`.

There are also GNU extension functions `clog10`, `clog10f` and `clog10l` which names are reserved by ISO C99 for future use. All these functions have 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=c90`).

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`, `memchr`, `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. In the same fashion, GCC provides `fpclassify`, `isfinite`, `isinf_sign`, `isnormal` and `signbit` built-ins used with `__builtin_` prefixed. The `isinf` and `isnan` built-in functions appear both with and without the `__builtin_` prefix.

Built-in Function: *void* ***__builtin_alloca** *(size_t size)*

> The `__builtin_alloca` function must be called at block scope. The function allocates an object *size* bytes large on the stack of the calling function. The object is aligned on the default stack alignment boundary for the target determined by the `__BIGGEST_ALIGNMENT__` macro. The `__builtin_alloca` function returns a pointer to the first byte of the allocated object. The lifetime of the allocated object ends just before the calling function returns to its caller. This is so even when `__builtin_alloca` is called within a nested block.
>
> For example, the following function allocates eight objects of `n` bytes each on the stack, storing a pointer to each in consecutive elements of the array `a`. It then passes the array to function `g` which can safely use the storage pointed to by each of the array elements.

```
void f (unsigned n)
{
  void *a [8];
  for (int i = 0; i != 8; ++i)
    a [i] = __builtin_alloca (n);

  g (a, n);   // safe
}
```

Since the `__builtin_alloca` function doesn't validate its argument it is the responsibility of its caller to make sure the argument doesn't cause it to exceed the stack size limit. The `__builtin_alloca` function is provided to make it possible to allocate on the stack arrays of bytes with an upper bound that may be computed at run time. Since C99 Variable Length Arrays offer similar functionality under a portable, more convenient, and safer interface they are recommended instead, in both C99 and C++ programs where GCC provides them as an extension. See Variable Length, for details.

Built-in Function: *void* ***__builtin_alloca_with_align*** *(size_t size, size_t alignment)*

The `__builtin_alloca_with_align` function must be called at block scope. The function allocates an object *size* bytes large on the stack of the calling function. The allocated object is aligned on the boundary specified by the argument *alignment* whose unit is given in bits (not bytes). The *size* argument must be positive and not exceed the stack size limit. The *alignment* argument must be a constant integer expression that evaluates to a power of 2 greater than or equal to `CHAR_BIT` and less than some unspecified maximum. Invocations with other values are rejected with an error indicating the valid bounds. The function returns a pointer to the first byte of the allocated object. The lifetime of the allocated object ends at the end of the block in which the function was called. The allocated storage is released no later than just before the calling function returns to its caller, but may be released at the end of the block in which the function was called.

For example, in the following function the call to `g` is unsafe because when `overalign` is non-zero, the space allocated by `__builtin_alloca_with_align` may have been released at the end of the `if` statement in which it was called.

```
  void f (unsigned n, bool overalign)
  {
    void *p;
    if (overalign)
      p = __builtin_alloca_with_align (n, 64 /* bits */);
    else
      p = __builtin_alloc (n);

    g (p, n);   // unsafe
  }
```

Since the `__builtin_alloca_with_align` function doesn't validate its *size* argument it is the responsibility of its caller to make sure the argument doesn't cause it to exceed the stack size limit. The `__builtin_alloca_with_align` function is provided to make it possible to allocate on the stack overaligned arrays of bytes with an upper bound that may be computed at run time. Since C99 Variable Length Arrays offer the same functionality under a portable, more convenient, and safer interface they are recommended instead, in both C99 and C++ programs where GCC provides them as an extension. See Variable Length, for details.

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

```
#define foo(x)                                                  \
  ({                                                           \
    typeof (x) tmp = (x);                                       \
    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_call_with_static_chain** *(call_exp, pointer_exp)*

The *call_exp* expression must be a function call, and the *pointer_exp* expression must be a pointer. The *pointer_exp* is passed to the function call in the target's static chain location. The result of builtin is the result of the function call.

Note: This builtin is only available for C. This builtin can be used to call Go closures from 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 an integer constant expression, is nonzero. Otherwise it returns *exp2*.

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 is 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: *type* **__builtin_complex** *(real, imag)*

The built-in function `__builtin_complex` is provided for use in implementing the ISO C11 macros `CMPLXF`, `CMPLX` and `CMPLXL`. *real* and *imag* must have the same type, a real binary floating-point type, and the result has the corresponding complex type with real and imaginary parts *real* and *imag*. Unlike '`real + I * imag`', this works even when infinities, NaNs and negative zeros are involved.

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 typically use this function in an embedded application where memory is 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 never returns 1 when you call the inline function with a string constant or compound literal (see Compound Literals) and does 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, including the case where `__builtin_constant_p` returns 1 because *EXPRESSION* can be folded to a constant but *EXPRESSION* contains operands that are not otherwise permitted in a static initializer (for example, `0 && foo ()`). GCC must be more conservative about evaluating the built-in in this case, because it has no opportunity to perform optimization.

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 semantics of the built-in are that it is expected that *exp* == *c*. For example:

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

indicates 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))
  foo (*ptr);
```

when testing pointer or floating-point values.

Built-in Function: *void* **__builtin_trap** *(void)*

This function causes the program to exit abnormally. GCC implements this function by using a target-dependent mechanism (such as intentionally executing an illegal instruction) or by calling `abort`. The mechanism used may vary from release to release so you should not rely on any particular implementation.

Built-in Function: *void* **__builtin_unreachable** *(void)*

If control flow reaches the point of the `__builtin_unreachable`, the program is undefined. It is useful in situations where the compiler cannot deduce the unreachability of the code.

One such case is immediately following an `asm` statement that either never terminates, or one that transfers control elsewhere and never returns. In this example, without the `__builtin_unreachable`, GCC issues a warning that control reaches the end of a non-void function. It also generates code to return after the `asm`.

```
int f (int c, int v)
{
  if (c)
    {
      return v;
    }
  else
    {
      asm("jmp error_handler");
```

```
          __builtin_unreachable ();
        }
    }
```

Because the `asm` statement unconditionally transfers control out of the function, control never reaches the end of the function body. The `__builtin_unreachable` is in fact unreachable and communicates this fact to the compiler.

Another use for `__builtin_unreachable` is following a call a function that never returns but that is not declared `__attribute__((noreturn))`, as in this example:

```
void function_that_never_returns (void);

int g (int c)
{
  if (c)
    {
      return 1;
    }
  else
    {
      function_that_never_returns ();
      __builtin_unreachable ();
    }
}
```

Built-in Function: *void ** **__builtin_assume_aligned** *(const void *exp, size_t align, ...)*

This function returns its first argument, and allows the compiler to assume that the returned pointer is at least *align* bytes aligned. This built-in can have either two or three arguments, if it has three, the third argument should have integer type, and if it is nonzero means misalignment offset. For example:

```
void *x = __builtin_assume_aligned (arg, 16);
```

means that the compiler can assume `x`, set to `arg`, is at least 16-byte aligned, while:

```
void *x = __builtin_assume_aligned (arg, 32, 8);
```

means that the compiler can assume for `x`, set to `arg`, that `(char *) x - 8` is 32-byte aligned.

Built-in Function: *int* **__builtin_LINE** *()*

This function is the equivalent of the preprocessor `__LINE__` macro and returns a constant integer expression that evaluates to the line number of the invocation of the built-in. When used as a C++ default argument for a function *F*, it returns the line number of the call to *F*.

Built-in Function: *const char ** **__builtin_FUNCTION** *()*

This function is the equivalent of the `__FUNCTION__` symbol and returns an address constant pointing to the name of the function from which the built-in was invoked, or the empty string if the invocation is not at function scope. When used as a C++ default argument for a function *F*, it returns the name of *F*'s caller or the empty string if the call was not made at function scope.

Built-in Function: *const char ** **__builtin_FILE** *()*

This function is the equivalent of the preprocessor `__FILE__` macro and returns an address constant pointing to the file name containing the invocation of the built-in, or the empty string if the invocation is not at function scope. When used as a C++ default argument for a function *F*, it returns the file name of the call to *F* or the empty string if the call was not made at function scope.

For example, in the following, each call to function `foo` will print a line similar to `"file.c:123: foo: message"` with the name of the file and the line number of the `printf` call, the name of the function `foo`, followed by the word `message`.

```
const char*
function (const char *func = __builtin_FUNCTION ())
{
  return func;
}

void foo (void)
{
  printf ("%s:%i: %s: message\n", file (), line (), function ());
}
```

Built-in Function: *void* **__builtin___clear_cache** *(char *begin, char *end)*

This function is used to flush the processor's instruction cache for the region of memory between *begin* inclusive and *end* exclusive. Some targets require that the instruction cache be flushed, after modifying memory containing code, in order to obtain deterministic behavior.

If the target does not require instruction cache flushes, `__builtin___clear_cache` has no effect. Otherwise either instructions are emitted in-line to clear the instruction cache or a call to the `__clear_cache` function in libgcc is made.

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 are 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` does not fault if `p->next` is not a valid address, but evaluation faults 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: *_Floatn* **__builtin_huge_valf*n*** *(void)*

Similar to `__builtin_huge_val`, except the return type is `_Float`*n*.

Built-in Function: *_Floatnx* **__builtin_huge_valf*n*x** *(void)*

Similar to `__builtin_huge_val`, except the return type is `_Float`*n*`x`.

Built-in Function: *int* **__builtin_fpclassify** *(int, int, int, int, int, ...)*

This built-in implements the C99 fpclassify functionality. The first five int arguments should be the target library's notion of the possible FP classes and are used for return values. They must be constant values and they must appear in this order: `FP_NAN`, `FP_INFINITE`, `FP_NORMAL`, `FP_SUBNORMAL` and `FP_ZERO`. The ellipsis is for exactly one floating-point value to classify. GCC treats the last argument as type-generic, which means it does not do default promotion from float to 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: *_Decimal32* **__builtin_infd32** *(void)*

Similar to `__builtin_inf`, except the return type is `_Decimal32`.

Built-in Function: *_Decimal64* **__builtin_infd64** *(void)*

Similar to `__builtin_inf`, except the return type is `_Decimal64`.

Built-in Function: *_Decimal128* **__builtin_infd128** *(void)*

Similar to `__builtin_inf`, except the return type is `_Decimal128`.

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: *_Floatn* **__builtin_inff*n*** *(void)*

Similar to `__builtin_inf`, except the return type is `_Floatn`.

Built-in Function: *_Floatn* **__builtin_inffnx** *(void)*

Similar to `__builtin_inf`, except the return type is `_Floatnx`.

Built-in Function: *int* **__builtin_isinf_sign** *(...)*

Similar to `isinf`, except the return value is -1 for an argument of `-Inf` and 1 for an argument of `+Inf`. Note while the parameter list is an ellipsis, this function only accepts exactly one floating-point argument. GCC treats this parameter as type-generic, which means it does not do default promotion from float to 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 all of which would have been consumed by `strtol`, is evaluated early enough that it is considered a compile-time constant.

Built-in Function: *_Decimal32* **__builtin_nand32** *(const char *str)*

Similar to `__builtin_nan`, except the return type is `_Decimal32`.

Built-in Function: *_Decimal64* **__builtin_nand64** *(const char *str)*

Similar to `__builtin_nan`, except the return type is `_Decimal64`.

Built-in Function: *_Decimal128* **__builtin_nand128** *(const char *str)*

Similar to `__builtin_nan`, except the return type is `_Decimal128`.

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: *_Floatn* **__builtin_nanf*n*** *(const char *str)*

Similar to `__builtin_nan`, except the return type is `_Float`*n*.

Built-in Function: *_Floatnx* **__builtin_nanf*n*x** *(const char *str)*

Similar to `__builtin_nan`, except the return type is `_Float`*n*`x`.

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: *_Floatn* **__builtin_nansf*n*** *(const char *str)*

Similar to `__builtin_nans`, except the return type is `_Float`*n*.

Built-in Function: *_Floatnx* **__builtin_nansf*n*x** *(const char *str)*

Similar to `__builtin_nans`, except the return type is `_Float`*n*`x`.

Built-in Function: *int* **__builtin_ffs** *(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_clrsb** *(int x)*

Returns the number of leading redundant sign bits in *x*, i.e. the number of bits following the most significant bit that are identical to it. There are no special cases for 0 or other values.

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** *(long)*

Similar to `__builtin_ffs`, except the argument type is `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_clrsbl** *(long)*

Similar to `__builtin_clrsb`, except the argument type is `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** *(long long)*

Similar to `__builtin_ffs`, except the argument type is `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_clrsbll** *(long long)*

Similar to `__builtin_clrsb`, except the argument type is `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`.

Built-in Function: *uint16_t* **__builtin_bswap16** *(uint16_t x)*

Returns *x* with the order of the bytes reversed; for example, `0xaabb` becomes `0xbbaa`. Byte here always means exactly 8 bits.

Built-in Function: *uint32_t* **__builtin_bswap32** *(uint32_t x)*

Similar to `__builtin_bswap16`, except the argument and return types are 32 bit.

Built-in Function: *uint64_t* **__builtin_bswap64** *(uint64_t x)*

Similar to `__builtin_bswap32`, except the argument and return types are 64 bit.
