GNU libiberty

Release 13.0.0 (experimental 20221108)

Phil Edwards et al.

Nov 10, 2022

CONTENTS

CHAPTER

ONE

COPYRIGHT

CHAPTER

TWO

INTRODUCTION

The `libiberty` library is a collection of subroutines used by various GNU programs. It is available under the Library General Public License; for more information, see Library Copying.

CHAPTER THREE

USING

To date, `libiberty` is generally not installed on its own. It has evolved over years but does not have its own version number nor release schedule.

Possibly the easiest way to use `libiberty` in your projects is to drop the `libiberty` code into your project's sources, and to build the library along with your own sources; the library would then be linked in at the end. This prevents any possible version mismatches with other copies of libiberty elsewhere on the system.

Passing `--enable-install-libiberty` to the **`configure`** script when building `libiberty` causes the header files and archive library to be installed when make install is run. This option also takes an (optional) argument to specify the installation location, in the same manner as `--prefix`.

For your own projects, an approach which offers stability and flexibility is to include `libiberty` with your code, but allow the end user to optionally choose to use a previously-installed version instead. In this way the user may choose (for example) to install `libiberty` as part of GCC, and use that version for all software built with that compiler. (This approach has proven useful with software using the GNU `readline` library.)

Making use of `libiberty` code usually requires that you include one or more header files from the `libiberty` distribution. (They will be named as necessary in the function descriptions.) At link time, you will need to add `-liberty` to your link command invocation.

CHAPTER

FOUR

OVERVIEW

Functions contained in `libiberty` can be divided into three general categories.

4.1 Supplemental Functions

Certain operating systems do not provide functions which have since become standardized, or at least common. For example, the Single Unix Specification Version 2 requires that the `basename` function be provided, but an OS which predates that specification might not have this function. This should not prevent well-written code from running on such a system.

Similarly, some functions exist only among a particular 'flavor' or 'family' of operating systems. As an example, the `bzero` function is often not present on systems outside the BSD-derived family of systems.

Many such functions are provided in `libiberty`. They are quickly listed here with little description, as systems which lack them become less and less common. Each function *`foo`* is implemented in *`foo`*`.c` but not declared in any `libiberty` header file; more comments and caveats for each function's implementation are often available in the source file. Generally, the function can simply be declared as `extern`.

4.2 Replacement Functions

Some functions have extremely limited implementations on different platforms. Other functions are tedious to use correctly; for example, proper use of `malloc` calls for the return value to be checked and appropriate action taken if memory has been exhausted. A group of 'replacement functions' is available in `libiberty` to address these issues for some of the most commonly used subroutines.

All of these functions are declared in the `libiberty.h` header file. Many of the implementations will use preprocessor macros set by GNU Autoconf, if you decide to make use of that program. Some of these functions may call one another.

4.2.1 Memory Allocation

The functions beginning with the letter `x` are wrappers around standard functions; the functions provided by the system environment are called and their results checked before the results are passed back to client code. If the standard functions fail, these wrappers will terminate the program. Thus, these versions can be used with impunity.

4.2.2 Exit Handlers

The existence and implementation of the `atexit` routine varies amongst the flavors of Unix. `libiberty` provides an unvarying dependable implementation via `xatexit` and `xexit`.

4.2.3 Error Reporting

These are a set of routines to facilitate programming with the system `errno` interface. The `libiberty` source file `strerror.c` contains a good deal of documentation for these functions.

4.3 Extensions

`libiberty` includes additional functionality above and beyond standard functions, which has proven generically useful in GNU programs, such as obstacks and regex. These functions are often copied from other projects as they gain popularity, and are included here to provide a central location from which to use, maintain, and distribute them.

4.3.1 Obstacks

An *obstack* is a pool of memory containing a stack of objects. You can create any number of separate obstacks, and then allocate objects in specified obstacks. Within each obstack, the last object allocated must always be the first one freed, but distinct obstacks are independent of each other.

Aside from this one constraint of order of freeing, obstacks are totally general: an obstack can contain any number of objects of any size. They are implemented with macros, so allocation is usually very fast as long as the objects are usually small. And the only space overhead per object is the padding needed to start each object on a suitable boundary.

Creating Obstacks

The utilities for manipulating obstacks are declared in the header file `obstack.h`.

Data Type struct obstackAn obstack is represented by a data structure of type `struct obstack`. This structure has a small fixed size; it records the status of the obstack and how to find the space in which objects are allocated. It does not contain any of the objects themselves. You should not try to access the contents of the structure directly; use only the macros described in this chapter.

You can declare variables of type `struct obstack` and use them as obstacks, or you can allocate obstacks dynamically like any other kind of object. Dynamic allocation of obstacks allows your program to have a variable number of different stacks. (You can even allocate an obstack structure in another obstack, but this is rarely useful.)

All the macros that work with obstacks require you to specify which obstack to use. You do this with a pointer of type `struct obstack *`. In the following, we often say 'an obstack' when strictly speaking the object at hand is such a pointer.

The objects in the obstack are packed into large blocks called *chunks*. The `struct obstack` structure points to a chain of the chunks currently in use.

The obstack library obtains a new chunk whenever you allocate an object that won't fit in the previous chunk. Since the obstack library manages chunks automatically, you don't need to pay much attention to them, but you do need to supply a function which the obstack library should use to get a chunk. Usually you supply a function which uses `malloc` directly or indirectly. You must also supply a function to free a chunk. These matters are described in the following section.

Preparing for Using Obstacks

Each source file in which you plan to use obstacks must include the header file `obstack.h`, like this:

```
#include <obstack.h>
```

Also, if the source file uses the macro `obstack_init`, it must declare or define two macros that will be called by the obstack library. One, `obstack_chunk_alloc`, is used to allocate the chunks of memory into which objects are packed. The other, `obstack_chunk_free`, is used to return chunks when the objects in them are freed. These macros should appear before any use of obstacks in the source file.

Usually these are defined to use `malloc` via the intermediary `xmalloc` (see Unconstrained Allocation in The GNU C Library Reference Manual). This is done with the following pair of macro definitions:

```
#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free
```

Though the memory you get using obstacks really comes from `malloc`, using obstacks is faster because `malloc` is called less often, for larger blocks of memory. See Obstack Chunks, for full details.

At run time, before the program can use a `struct obstack` object as an obstack, it must initialize the obstack by calling `obstack_init` or one of its variants, `obstack_begin`, `obstack_specify_allocation`, or `obstack_specify_allocation_with_arg`.

int **obstack_init**(struct obstack *obstack_ptr)

Initialize obstack *obstack_ptr* for allocation of objects. This macro calls the obstack's `obstack_chunk_alloc` function. If allocation of memory fails, the function pointed to by `obstack_alloc_failed_handler` is called. The `obstack_init` macro always returns 1 (Compatibility notice: Former versions of obstack returned 0 if allocation failed).

Here are two examples of how to allocate the space for an obstack and initialize it. First, an obstack that is a static variable:

```
static struct obstack myobstack;
...
obstack_init (&myobstack);
```

Second, an obstack that is itself dynamically allocated:

```
struct obstack *myobstack_ptr
  = (struct obstack *) xmalloc (sizeof (struct obstack));

obstack_init (myobstack_ptr);
```

int **obstack_begin**(struct obstack *obstack_ptr, size_t chunk_size)

Like `obstack_init`, but specify chunks to be at least *chunk_size* bytes in size.

int **obstack_specify_allocation**(struct obstack *obstack_ptr, size_t chunk_size, size_t alignment, void *(*chunkfun)(size_t), void (*freefun)(void*))

Like `obstack_init`, specifying chunk size, chunk alignment, and memory allocation functions. A *chunk_size* or *alignment* of zero results in the default size or alignment respectively being used.

int **obstack_specify_allocation_with_arg**(struct obstack *obstack_ptr, size_t chunk_size, size_t alignment, void *(*chunkfun)(void*, size_t), void (*freefun)(void*, void*), void *arg)

Like `obstack_specify_allocation`, but specifying memory allocation functions that take an extra first argument, *arg*.

Variable obstack_alloc_failed_handlerThe value of this variable is a pointer to a function that `obstack` uses when `obstack_chunk_alloc` fails to allocate memory. The default action is to print a message and abort. You should supply a function that either calls `exit` (see Program Termination in The GNU C Library Reference Manual) or `longjmp` and doesn't return.

```
void my_obstack_alloc_failed (void)
...
obstack_alloc_failed_handler = &my_obstack_alloc_failed;
```

Allocation in an Obstack

The most direct way to allocate an object in an obstack is with `obstack_alloc`, which is invoked almost like `malloc`.

void ***obstack_alloc**(struct obstack *obstack_ptr, size_t size)

This allocates an uninitialized block of *size* bytes in an obstack and returns its address. Here *obstack_ptr* specifies which obstack to allocate the block in; it is the address of the `struct obstack` object which represents the obstack. Each obstack macro requires you to specify an *obstack_ptr* as the first argument.

This macro calls the obstack's `obstack_chunk_alloc` function if it needs to allocate a new chunk of memory; it calls `obstack_alloc_failed_handler` if allocation of memory by `obstack_chunk_alloc` failed.

For example, here is a function that allocates a copy of a string *str* in a specific obstack, which is in the variable `string_obstack` :

```
struct obstack string_obstack;

char *
copystring (char *string)
{
  size_t len = strlen (string) + 1;
  char *s = (char *) obstack_alloc (&string_obstack, len);
  memcpy (s, string, len);
  return s;
}
```

To allocate a block with specified contents, use the macro `obstack_copy`.

void ***obstack_copy**(struct obstack *obstack_ptr, void *address, size_t size)

This allocates a block and initializes it by copying *size* bytes of data starting at *address*. It calls `obstack_alloc_failed_handler` if allocation of memory by `obstack_chunk_alloc` failed.

void ***obstack_copy0**(struct obstack *obstack_ptr, void *address, size_t size)

Like `obstack_copy`, but appends an extra byte containing a null character. This extra byte is not counted in the argument *size*.

The `obstack_copy0` macro is convenient for copying a sequence of characters into an obstack as a null-terminated string. Here is an example of its use:

```
char *
obstack_savestring (char *addr, size_t size)
{
  return obstack_copy0 (&myobstack, addr, size);
}
```

Contrast this with the previous example of `savestring` using `malloc` see (Basic Allocation).

Freeing Objects in an Obstack

To free an object allocated in an obstack, use the macro `obstack_free`. Since the obstack is a stack of objects, freeing one object automatically frees all other objects allocated more recently in the same obstack.

void **obstack_free**(struct obstack *obstack_ptr, void *object)

If *object* is a null pointer, everything allocated in the obstack is freed. Otherwise, *object* must be the address of an object allocated in the obstack. Then *object* is freed, along with everything allocated in *obstack* since *object*.

Note that if *object* is a null pointer, the result is an uninitialized obstack. To free all memory in an obstack but leave it valid for further allocation, call `obstack_free` with the address of the first object allocated on the obstack:

```
obstack_free (obstack_ptr, first_object_allocated_ptr);
```

Recall that the objects in an obstack are grouped into chunks. When all the objects in a chunk become free, the obstack library automatically frees the chunk (see Preparing for Using Obstacks). Then other obstacks, or non-obstack allocation, can reuse the space of the chunk.

Obstack Functions and Macros

The interfaces for using obstacks are shown here as functions to specify the return type and argument types, but they are really defined as macros. This means that the arguments don't actually have types, but they generally behave as if they have the types shown. You can call these macros like functions, but you cannot use them in any other way (for example, you cannot take their address).

Calling the macros requires a special precaution: namely, the first operand (the obstack pointer) may not contain any side effects, because it may be computed more than once. For example, if you write this:

```
obstack_alloc (get_obstack (), 4);
```

you will find that `get_obstack` may be called several times. If you use `*obstack_list_ptr++` as the obstack pointer argument, you will get very strange results since the incrementation may occur several times.

If you use the GNU C compiler, this precaution is not necessary, because various language extensions in GNU C permit defining the macros so as to compute each argument only once.

Note that arguments other than the first will only be evaluated once, even when not using GNU C.

`obstack.h` does declare a number of functions, `_obstack_begin`, `_obstack_begin_1`, `_obstack_newchunk`, `_obstack_free`, and `_obstack_memory_used`. You should not call these directly.

Growing Objects

Because memory in obstack chunks is used sequentially, it is possible to build up an object step by step, adding one or more bytes at a time to the end of the object. With this technique, you do not need to know how much data you will put in the object until you come to the end of it. We call this the technique of *growing objects*. The special macros for adding data to the growing object are described in this section.

You don't need to do anything special when you start to grow an object. Using one of the macros to add data to the object automatically starts it. However, it is necessary to say explicitly when the object is finished. This is done with `obstack_finish`.

The actual address of the object thus built up is not known until the object is finished. Until then, it always remains possible that you will add so much data that the object must be copied into a new chunk.

While the obstack is in use for a growing object, you cannot use it for ordinary allocation of another object. If you try to do so, the space already added to the growing object will become part of the other object.

void **obstack_blank**(struct obstack *obstack__ptr, size__t size)

> The most basic macro for adding to a growing object is `obstack_blank`, which adds space without initializing it.

void **obstack_grow**(struct obstack *obstack__ptr, void *data, size__t size)

> To add a block of initialized space, use `obstack_grow`, which is the growing-object analogue of `obstack_copy`. It adds *size* bytes of data to the growing object, copying the contents from *data*.

void **obstack_grow0**(struct obstack *obstack__ptr, void *data, size__t size)

> This is the growing-object analogue of `obstack_copy0`. It adds *size* bytes copied from *data*, followed by an additional null character.

void **obstack_1grow**(struct obstack *obstack__ptr, char c)

> To add one character at a time, use `obstack_1grow`. It adds a single byte containing *c* to the growing object.

void **obstack_ptr_grow**(struct obstack *obstack__ptr, void *data)

> Adding the value of a pointer one can use `obstack_ptr_grow`. It adds `sizeof (void *)` bytes containing the value of *data*.

void **obstack_int_grow**(struct obstack *obstack__ptr, int data)

> A single value of type `int` can be added by using `obstack_int_grow`. It adds `sizeof (int)` bytes to the growing object and initializes them with the value of *data*.

void ***obstack_finish**(struct obstack *obstack__ptr)

> When you are finished growing the object, use `obstack_finish` to close it off and return its final address.
>
> Once you have finished the object, the obstack is available for ordinary allocation or for growing another object.

When you build an object by growing it, you will probably need to know afterward how long it became. You need not keep track of this as you grow the object, because you can find out the length from the obstack with `obstack_object_size`, before finishing the object.

size_t **`obstack_object_size`**(struct obstack *obstack_ptr)

This macro returns the current size of the growing object, in bytes. Remember to call `obstack_object_size` *before* finishing the object. After it is finished, `obstack_object_size` will return zero.

If you have started growing an object and wish to cancel it, you should finish it and then free it, like this:

```
obstack_free (obstack_ptr, obstack_finish (obstack_ptr));
```

This has no effect if no object was growing.

Extra Fast Growing Objects

The usual macros for growing objects incur overhead for checking whether there is room for the new growth in the current chunk. If you are frequently constructing objects in small steps of growth, this overhead can be significant.

You can reduce the overhead by using special 'fast growth' macros that grow the object without checking. In order to have a robust program, you must do the checking yourself. If you do this checking in the simplest way each time you are about to add data to the object, you have not saved anything, because that is what the ordinary growth macros do. But if you can arrange to check less often, or check more efficiently, then you make the program faster.

`obstack_room` returns the amount of room available in the current chunk.

size_t **`obstack_room`**(struct obstack *obstack_ptr)

This returns the number of bytes that can be added safely to the current growing object (or to an object about to be started) in obstack *`obstack`* using the fast growth macros.

While you know there is room, you can use these fast growth macros for adding data to a growing object:

void **`obstack_1grow_fast`**(struct obstack *obstack_ptr, char c)

`obstack_1grow_fast` adds one byte containing the character *`c`* to the growing object in obstack *`obstack_ptr`*.

void **`obstack_ptr_grow_fast`**(struct obstack *obstack_ptr, void *data)

`obstack_ptr_grow_fast` adds `sizeof (void *)` bytes containing the value of *`data`* to the growing object in obstack *`obstack_ptr`*.

void **`obstack_int_grow_fast`**(struct obstack *obstack_ptr, int data)

`obstack_int_grow_fast` adds `sizeof (int)` bytes containing the value of *`data`* to the growing object in obstack *`obstack_ptr`*.

void **obstack_blank_fast**(struct obstack *obstack_ptr, size_t size)

> `obstack_blank_fast` adds *size* bytes to the growing object in obstack *obstack_ptr* without initializing them.

When you check for space using `obstack_room` and there is not enough room for what you want to add, the fast growth macros are not safe. In this case, simply use the corresponding ordinary growth macro instead. Very soon this will copy the object to a new chunk; then there will be lots of room available again.

So, each time you use an ordinary growth macro, check afterward for sufficient space using `obstack_room`. Once the object is copied to a new chunk, there will be plenty of space again, so the program will start using the fast growth macros again.

Here is an example:

```
void
add_string (struct obstack *obstack, const char *ptr, size_t len)
{
  while (len > 0)
    {
      size_t room = obstack_room (obstack);
      if (room == 0)
        {
          /* Not enough room.  Add one character slowly,
             which may copy to a new chunk and make room.  */
          obstack_1grow (obstack, *ptr++);
          len--;
        }
      else
        {
          if (room > len)
            room = len;
          /* Add fast as much as we have room for. */
          len -= room;
          while (room-- > 0)
            obstack_1grow_fast (obstack, *ptr++);
        }
    }
}
```

You can use `obstack_blank_fast` with a 'negative' size argument to make the current object smaller. Just don't try to shrink it beyond zero length—there's no telling what will happen if you do that. Earlier versions of obstacks allowed you to use `obstack_blank` to shrink objects. This will no longer work.

Status of an Obstack

Here are macros that provide information on the current status of allocation in an obstack. You can use them to learn about an object while still growing it.

void ***obstack_base**(struct obstack *obstack_ptr)

This macro returns the tentative address of the beginning of the currently growing object in *obstack_ptr*. If you finish the object immediately, it will have that address. If you make it larger first, it may outgrow the current chunk—then its address will change!

If no object is growing, this value says where the next object you allocate will start (once again assuming it fits in the current chunk).

void ***obstack_next_free**(struct obstack *obstack_ptr)

This macro returns the address of the first free byte in the current chunk of obstack *obstack_ptr*. This is the end of the currently growing object. If no object is growing, `obstack_next_free` returns the same value as `obstack_base`.

size_t **obstack_object_size**(struct obstack *obstack_ptr)

This macro returns the size in bytes of the currently growing object. This is equivalent to

```
((size_t) (obstack_next_free (obstack_ptr) - obstack_base (obstack_ptr)))
```

Alignment of Data in Obstacks

Each obstack has an *alignment boundary*; each object allocated in the obstack automatically starts on an address that is a multiple of the specified boundary. By default, this boundary is aligned so that the object can hold any type of data.

To access an obstack's alignment boundary, use the macro `obstack_alignment_mask`.

size_t **obstack_alignment_mask**(struct obstack *obstack_ptr)

The value is a bit mask; a bit that is 1 indicates that the corresponding bit in the address of an object should be 0. The mask value should be one less than a power of 2; the effect is that all object addresses are multiples of that power of 2. The default value of the mask is a value that allows aligned objects to hold any type of data: for example, if its value is 3, any type of data can be stored at locations whose addresses are multiples of 4. A mask value of 0 means an object can start on any multiple of 1 (that is, no alignment is required).

The expansion of the macro `obstack_alignment_mask` is an lvalue, so you can alter the mask by assignment. For example, this statement:

```
obstack_alignment_mask (obstack_ptr) = 0;
```

has the effect of turning off alignment processing in the specified obstack.

Note that a change in alignment mask does not take effect until *after* the next time an object is allocated or finished in the obstack. If you are not growing an object, you can make the new alignment mask take effect immediately by calling `obstack_finish`. This will finish a zero-length object and then do proper alignment for the next object.

Obstack Chunks

Obstacks work by allocating space for themselves in large chunks, and then parceling out space in the chunks to satisfy your requests. Chunks are normally 4096 bytes long unless you specify a different chunk size. The chunk size includes 8 bytes of overhead that are not actually used for storing objects. Regardless of the specified size, longer chunks will be allocated when necessary for long objects.

The obstack library allocates chunks by calling the function `obstack_chunk_alloc`, which you must define. When a chunk is no longer needed because you have freed all the objects in it, the obstack library frees the chunk by calling `obstack_chunk_free`, which you must also define.

These two must be defined (as macros) or declared (as functions) in each source file that uses `obstack_init` (see Creating Obstacks). Most often they are defined as macros like this:

```
#define obstack_chunk_alloc malloc
#define obstack_chunk_free free
```

Note that these are simple macros (no arguments). Macro definitions with arguments will not work! It is necessary that `obstack_chunk_alloc` or `obstack_chunk_free`, alone, expand into a function name if it is not itself a function name.

If you allocate chunks with `malloc`, the chunk size should be a power of 2. The default chunk size, 4096, was chosen because it is long enough to satisfy many typical requests on the obstack yet short enough not to waste too much memory in the portion of the last chunk not yet used.

size_t **obstack_chunk_size**(struct obstack *obstack_ptr)

This returns the chunk size of the given obstack.

Since this macro expands to an lvalue, you can specify a new chunk size by assigning it a new value. Doing so does not affect the chunks already allocated, but will change the size of chunks allocated for that particular obstack in the future. It is unlikely to be useful to make the chunk size smaller, but making it larger might improve efficiency if you are allocating many objects whose size is comparable to the chunk size. Here is how to do so cleanly:

```
if (obstack_chunk_size (obstack_ptr) < new-chunk-size)
  obstack_chunk_size (obstack_ptr) = new-chunk-size;
```

Summary of Obstack Macros

Here is a summary of all the macros associated with obstacks. Each takes the address of an obstack (`struct obstack *`) as its first argument.

int **obstack_init**(struct obstack *obstack_ptr)

Initialize use of an obstack. See Creating Obstacks.

int **obstack_begin**(struct obstack *obstack_ptr, size_t chunk_size)

Initialize use of an obstack, with an initial chunk of *chunk_size* bytes.

int **obstack_specify_allocation**(struct obstack *obstack_ptr, size_t chunk_size, size_t alignment, void *(*chunkfun)(size_t), void (*freefun)(void*))

Initialize use of an obstack, specifying intial chunk size, chunk alignment, and memory allocation functions.

int **obstack_specify_allocation_with_arg**(struct obstack *obstack_ptr, size_t chunk_size, size_t alignment, void *(*chunkfun)(void*, size_t), void (*freefun)(void*, void*), void *arg)

Like `obstack_specify_allocation`, but specifying memory allocation functions that take an extra first argument, *arg*.

void ***obstack_alloc**(struct obstack *obstack_ptr, size_t size)

Allocate an object of *size* uninitialized bytes. See Allocation in an Obstack.

void ***obstack_copy**(struct obstack *obstack_ptr, void *address, size_t size)

Allocate an object of *size* bytes, with contents copied from *address*. See Allocation in an Obstack.

void ***obstack_copy0**(struct obstack *obstack_ptr, void *address, size_t size)

Allocate an object of *size* +1 bytes, with *size* of them copied from *address*, followed by a null character at the end. See Allocation in an Obstack.

void **obstack_free**(struct obstack *obstack_ptr, void *object)

Free *object* (and everything allocated in the specified obstack more recently than *object*). See Freeing Objects in an Obstack.

void **obstack_blank**(struct obstack *obstack_ptr, size_t size)

Add *size* uninitialized bytes to a growing object. See Growing Objects.

void **obstack_grow**(struct obstack *obstack_ptr, void *address, size_t size)

Add *size* bytes, copied from *address*, to a growing object. See Growing Objects.

void **obstack_grow0**(struct obstack *obstack_ptr, void *address, size_t size)

Add *size* bytes, copied from *address*, to a growing object, and then add another byte containing a null character. See Growing Objects.

void **obstack_1grow**(struct obstack *obstack_ptr, char data_char)

Add one byte containing *data-char* to a growing object. See Growing Objects.

void ***obstack_finish**(struct obstack *obstack_ptr)

Finalize the object that is growing and return its permanent address. See Growing Objects.

size_t **obstack_object_size**(struct obstack *obstack_ptr)

Get the current size of the currently growing object. See Growing Objects.

void **obstack_blank_fast**(struct obstack *obstack_ptr, size_t size)

Add *size* uninitialized bytes to a growing object without checking that there is enough room. See Extra Fast Growing Objects.

void **obstack_1grow_fast**(struct obstack *obstack_ptr, char data_char)

Add one byte containing *data-char* to a growing object without checking that there is enough room. See Extra Fast Growing Objects.

size_t **obstack_room**(struct obstack *obstack_ptr)

Get the amount of room now available for growing the current object. See Extra Fast Growing Objects.

size_t **obstack_alignment_mask**(struct obstack *obstack_ptr)

The mask used for aligning the beginning of an object. This is an lvalue. See Alignment of Data in Obstacks.

size_t **obstack_chunk_size**(struct obstack *obstack_ptr)

The size for allocating chunks. This is an lvalue. See Obstack Chunks.

void ***obstack_base**(struct obstack *obstack_ptr)

Tentative starting address of the currently growing object. See Status of an Obstack.

void ***obstack_next_free**(struct obstack *obstack_ptr)

Address just after the end of the currently growing object. See Status of an Obstack.

CHAPTER FIVE

FUNCTION, VARIABLE, AND MACRO LISTING.

void *`alloca`(size_t size)

This function allocates memory which will be automatically reclaimed after the procedure exits. The `libiberty` implementation does not free the memory immediately but will do so eventually during subsequent calls to this function. Memory is allocated using `xmalloc` under normal circumstances.

The header file `alloca-conf.h` can be used in conjunction with the GNU Autoconf test `AC_FUNC_ALLOCA` to test for and properly make available this function. The `AC_FUNC_ALLOCA` test requires that client code use a block of preprocessor code to be safe (see the Autoconf manual for more); this header incorporates that logic and more, including the possibility of a GCC built-in function.

int `asprintf`(char **resptr, const char *format, ...)

Like `sprintf`, but instead of passing a pointer to a buffer, you pass a pointer to a pointer. This function will compute the size of the buffer needed, allocate memory with `malloc`, and store a pointer to the allocated memory in `*resptr`. The value returned is the same as `sprintf` would return. If memory could not be allocated, minus one is returned and `NULL` is stored in `*resptr`.

int `atexit`(void (*f)())

Causes function *f* to be called at exit. Returns 0.

char *`basename`(const char *name)

Returns a pointer to the last component of pathname *name*. Behavior is undefined if the pathname ends in a directory separator.

int `bcmp`(char *x, char *y, int count)

Compares the first *count* bytes of two areas of memory. Returns zero if they are the same, nonzero otherwise. Returns zero if *count* is zero. A nonzero result only indicates a difference, it does not indicate any sorting order (say, by having a positive result mean *x* sorts before *y*).

void `bcopy`(char *in, char *out, int length)

Copies *length* bytes from memory region *in* to region *out*. The use of `bcopy` is deprecated in new programs.

void *`bsearch`(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void*, const void*))

Performs a search over an array of *nmemb* elements pointed to by *base* for a member that matches the object pointed to by *key*. The size of each member is specified by *size*. The array contents should be sorted in ascending order according to the *compar* comparison function. This routine should take two arguments pointing to the *key* and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the *key* object is respectively less than, matching, or greater than the array member.

void ***bsearch_r**(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void*, const void*, void*), void *arg)

Performs a search over an array of *nmemb* elements pointed to by *base* for a member that matches the object pointed to by *key*. The size of each member is specified by *size*. The array contents should be sorted in ascending order according to the *compar* comparison function. This routine should take three arguments: the first two point to the *key* and to an array member, and the last is passed down unchanged from `bsearch_r` 's last argument. It should return an integer less than, equal to, or greater than zero if the *key* object is respectively less than, matching, or greater than the array member.

char ****buildargv**(char *sp)

Given a pointer to a string, parse the string extracting fields separated by whitespace and optionally enclosed within either single or double quotes (which are stripped off), and build a vector of pointers to copies of the string for each field. The input string remains unchanged. The last element of the vector is followed by a `NULL` element.

All of the memory for the pointer array and copies of the string is obtained from `xmalloc`. All of the memory can be returned to the system with the single function call `freeargv`, which takes the returned result of `buildargv`, as it's argument.

Returns a pointer to the argument vector if successful. Returns `NULL` if *sp* is `NULL` or if there is insufficient memory to complete building the argument vector.

If the input is a null string (as opposed to a `NULL` pointer), then buildarg returns an argument vector that has one arg, a null string.

void **bzero**(char *mem, int count)

Zeros *count* bytes starting at *mem*. Use of this function is deprecated in favor of `memset`.

void ***calloc**(size_t nelem, size_t elsize)

Uses `malloc` to allocate storage for *nelem* objects of *elsize* bytes each, then zeros the memory.

int **canonical_filename_eq**(const char *a, const char *b)

Return non-zero if file names *a* and *b* are equivalent. This function compares the canonical versions of the filenames as returned by `lrealpath()`, so that so that different file names pointing to the same underlying file are treated as being identical.

char ***choose_temp_base**(void)

Return a prefix for temporary file names or `NULL` if unable to find one. The current directory is chosen if all else fails so the program is exited if a temporary directory can't be found (`mktemp` fails). The buffer for the result is obtained with `xmalloc`.

This function is provided for backwards compatibility only. Its use is not recommended.

const char ***choose_tmpdir**()

Returns a pointer to a directory path suitable for creating temporary files in.

long **clock**(void)

Returns an approximation of the CPU time used by the process as a `clock_t` ; divide this number by `CLOCKS_PER_SEC` to get the number of seconds used.

char* concat (const char *s1, const char *s2, ..., NULL)

Concatenate zero or more of strings and return the result in freshly `xmalloc` ed memory. The argument list is terminated by the first `NULL` pointer encountered. Pointers to empty strings are ignored.

int **countargv**(char *const *argv)

Return the number of elements in *argv*. Returns zero if *argv* is NULL.

unsigned int **crc32**(const unsigned char *buf, int len, unsigned int init)

Compute the 32-bit CRC of *buf* which has length *len*. The starting value is *init* ; this may be used to compute the CRC of data split across multiple buffers by passing the return value of each call as the *init* parameter of the next.

This is used by the **gdb** remote protocol for the `qCRC` command. In order to get the same results as gdb for a block of data, you must pass the first CRC parameter as `0xffffffff`.

This CRC can be specified as:

Width
[32] Poly : 0x04c11db7 Init : parameter, typically 0xffffffff RefIn : false RefOut : false XorOut : 0

This differs from the "standard" CRC-32 algorithm in that the values are not reflected, and there is no final XOR value. These differences make it easy to compose the values of multiple blocks.

char ****dupargv**(char *const *vector)

Duplicate an argument vector. Simply scans through *vector*, duplicating each argument until the terminating `NULL` is found. Returns a pointer to the argument vector if successful. Returns `NULL` if there is insufficient memory to complete building the argument vector.

int **errno_max**(void)

Returns the maximum `errno` value for which a corresponding symbolic name or message is available. Note that in the case where we use the `sys_errlist` supplied by the system, it is possible for there to be more symbolic names than messages, or vice versa. In fact, the manual page for `perror(3C)` explicitly warns that one should check the size of the table (`sys_nerr`) before indexing it, since new error codes may be added to the system before they are added to the table. Thus `sys_nerr` might be smaller than value implied by the largest `errno` value defined in `<errno.h>`.

We return the maximum value that can be used to obtain a meaningful symbolic name or message.

void **expandargv**(int *argcp, char ***argvp)

The *argcp* and `argvp` arguments are pointers to the usual `argc` and `argv` arguments to `main`. This function looks for arguments that begin with the character `@`. Any such arguments are interpreted as 'response files'. The contents of the response file are interpreted as additional command line options. In particular, the file is separated into whitespace-separated strings;

each such string is taken as a command-line option. The new options are inserted in place of the option naming the response file, and `*argcp` and `*argvp` will be updated. If the value of `*argvp` is modified by this function, then the new value has been dynamically allocated and can be deallocated by the caller with `freeargv`. However, most callers will simply call `expandargv` near the beginning of `main` and allow the operating system to free the memory when the program exits.

int **fdmatch**(int fd1, int fd2)

Check to see if two open file descriptors refer to the same file. This is useful, for example, when we have an open file descriptor for an unnamed file, and the name of a file that we believe to correspond to that fd. This can happen when we are exec'd with an already open file (`stdout` for example) or from the SVR4 `/proc` calls that return open file descriptors for mapped address spaces. All we have to do is open the file by name and check the two file descriptors for a match, which is done by comparing major and minor device numbers and inode numbers.

FILE ***fdopen_unlocked**(int fildes, const char *mode)

Opens and returns a `FILE` pointer via `fdopen`. If the operating system supports it, ensure that the stream is setup to avoid any multi-threaded locking. Otherwise return the `FILE` pointer unchanged.

int **ffs**(int valu)

Find the first (least significant) bit set in *valu*. Bits are numbered from right to left, starting with bit 1 (corresponding to the value 1). If *valu* is zero, zero is returned.

int **filename_cmp**(const char *s1, const char *s2)

Return zero if the two file names *s1* and *s2* are equivalent. If not equivalent, the returned value is similar to what `strcmp` would return. In other words, it returns a negative value if *s1* is less than *s2*, or a positive value if *s2* is greater than *s2*.

This function does not normalize file names. As a result, this function will treat filenames that are spelled differently as different even in the case when the two filenames point to the same underlying file. However, it does handle the fact that on DOS-like file systems, forward and backward slashes are equal.

int **filename_eq**(const void *s1, const void *s2)

Return non-zero if file names *s1* and *s2* are equivalent. This function is for use with hashtab.c hash tables.

hashval_t **filename_hash**(const void *s)

Return the hash value for file name *s* that will be compared using filename_cmp. This function is for use with hashtab.c hash tables.

int **filename_ncmp**(const char *s1, const char *s2, size_t n)

Return zero if the two file names *s1* and *s2* are equivalent in range *n*. If not equivalent, the returned value is similar to what `strncmp` would return. In other words, it returns a negative value if *s1* is less than *s2*, or a positive value if *s2* is greater than *s2*.

This function does not normalize file names. As a result, this function will treat filenames that are spelled differently as different even in the case when the two filenames point to the same underlying file. However, it does handle the fact that on DOS-like file systems, forward and backward slashes are equal.

int **fnmatch**(const char *pattern, const char *string, int flags)

Matches *string* against *pattern*, returning zero if it matches, `FNM_NOMATCH` if not. *pattern* may contain the wildcards `?` to match any one character, `*` to match any zero or more characters, or a set of alternate characters in square brackets, like `[a-gt8]`, which match one character (`a` through `g`, or `t`, or `8`, in this example) if that one character is in the set. A set may be inverted (i.e., match anything except what's in the set) by giving `^` or `!` as the first character in the set. To include those characters in the set, list them as anything other than the first character of the set. To include a dash in the set, list it last in the set. A backslash character makes the following character not special, so for example you could match against a literal asterisk with `\*`. To match a literal backslash, use `\`.

`flags` controls various aspects of the matching process, and is a boolean OR of zero or more of the following values (defined in `<fnmatch.h>`):

FNM_PATHNAME

string is assumed to be a path name. No wildcard will ever match `/`.

FNM_NOESCAPE

Do not interpret backslashes as quoting the following special character.

FNM_PERIOD

A leading period (at the beginning of *string*, or if `FNM_PATHNAME` after a slash) is not matched by `*` or `?` but must be matched explicitly.

FNM_LEADING_DIR

Means that *string* also matches *pattern* if some initial part of *string* matches, and is followed by `/` and zero or more characters. For example, `foo*` would match either `foobar` or `foobar/grill`.

FNM_CASEFOLD

Ignores case when performing the comparison.

FILE ***fopen_unlocked**(const char *path, const char *mode)

Opens and returns a `FILE` pointer via `fopen`. If the operating system supports it, ensure that the stream is setup to avoid any multi-threaded locking. Otherwise return the `FILE` pointer unchanged.

void **freeargv**(char **vector)

Free an argument vector that was built using `buildargv`. Simply scans through *vector*, freeing the memory for each argument until the terminating `NULL` is found, and then frees *vector* itself.

FILE ***freopen_unlocked**(const char *path, const char *mode, FILE *stream)

Opens and returns a `FILE` pointer via `freopen`. If the operating system supports it, ensure that the stream is setup to avoid any multi-threaded locking. Otherwise return the `FILE` pointer unchanged.

long **get_run_time**(void)

Returns the time used so far, in microseconds. If possible, this is the time used by this process, else it is the elapsed time since the process started.

char ***getcwd**(char *pathname, int len)

Copy the absolute pathname for the current working directory into *pathname*, which is assumed to point to a buffer of at least *len* bytes, and return a pointer to the buffer. If the current directory's path doesn't fit in *len* characters, the result is `NULL` and `errno` is set. If *pathname* is a null pointer, `getcwd` will obtain *len* bytes of space using `malloc`.

int **getpagesize**(void)

Returns the number of bytes in a page of memory. This is the granularity of many of the system memory management routines. No guarantee is made as to whether or not it is the same as the basic memory management hardware page size.

char ***getpwd**(void)

Returns the current working directory. This implementation caches the result on the assumption that the process will not call `chdir` between calls to `getpwd`.

int **gettimeofday**(struct timeval *tp, void *tz)

Writes the current time to *tp*. This implementation requires that *tz* be NULL. Returns 0 on success, -1 on failure.

void **hex_init**(void)

Initializes the array mapping the current character set to corresponding hex values. This function must be called before any call to `hex_p` or `hex_value`. If you fail to call it, a default ASCII-based table will normally be used on ASCII systems.

int **hex_p**(int c)

Evaluates to non-zero if the given character is a valid hex character, or zero if it is not. Note that the value you pass will be cast to `unsigned char` within the macro.

unsigned int **hex_value**(int c)

Returns the numeric equivalent of the given character when interpreted as a hexadecimal digit. The result is undefined if you pass an invalid hex digit. Note that the value you pass will be cast to `unsigned char` within the macro.

The `hex_value` macro returns `unsigned int`, rather than signed `int`, to make it easier to use in parsing addresses from hex dump files: a signed `int` would be sign-extended when converted to a wider unsigned type — like `bfd_vma`, on some systems.

HOST_CHARSET

This macro indicates the basic character set and encoding used by the host: more precisely, the encoding used for character constants in preprocessor `#if` statements (the C "execution character set"). It is defined by `safe-ctype.h`, and will be an integer constant with one of the following values:

HOST_CHARSET_UNKNOWN

The host character set is unknown - that is, not one of the next two possibilities.

HOST_CHARSET_ASCII

The host character set is ASCII.

HOST_CHARSET_EBCDIC

The host character set is some variant of EBCDIC. (Only one of the nineteen EBCDIC varying characters is tested; exercise caution.)

htab_t **htab_create_typed_alloc**(size_t size, htab_hash hash_f, htab_eq eq_f, htab_del del_f, htab_alloc alloc_tab_f, htab_alloc alloc_f, htab_free free_f)

This function creates a hash table that uses two different allocators *alloc_tab_f* and *alloc_f* to use for allocating the table itself and its entries respectively. This is useful when variables of different types need to be allocated with different allocators.

The created hash table is slightly larger than *size* and it is initially empty (all the hash table entries are `HTAB_EMPTY_ENTRY`). The function returns the created hash table, or `NULL` if memory allocation fails.

char ***index**(char *s, int c)

Returns a pointer to the first occurrence of the character *c* in the string *s*, or `NULL` if not found. The use of `index` is deprecated in new programs in favor of `strchr`.

void **insque**(struct qelem *elem, struct qelem *pred)

void **remque**(struct qelem *elem)

Routines to manipulate queues built from doubly linked lists. The `insque` routine inserts *elem* in the queue immediately after *pred*. The `remque` routine removes *elem* from its containing queue. These routines expect to be passed pointers to structures which have as their first members a forward pointer and a back pointer, like this prototype (although no prototype is provided):

```
struct qelem {
  struct qelem *q_forw;
  struct qelem *q_back;
  char q_data[];
};
```

ISALPHA(c)

ISALNUM(c)

ISBLANK(c)

ISCNTRL(c)

ISDIGIT(c)

ISGRAPH(c)

ISLOWER(c)

ISPRINT(c)

ISPUNCT(c)

ISSPACE(c)

ISUPPER(c)

ISXDIGIT(c)

These twelve macros are defined by `safe-ctype.h`. Each has the same meaning as the corresponding macro (with name in lowercase) defined by the standard header `ctype.h`. For example, `ISALPHA` returns true for alphabetic characters and false for others. However, there are two differences between these macros and those provided by `ctype.h`:

- These macros are guaranteed to have well-defined behavior for all values representable by `signed char` and `unsigned char`, and for `EOF`.
- These macros ignore the current locale; they are true for these fixed sets of characters:

`ALPHA`	A-Za-z
`ALNUM`	A-Za-z0-9
`BLANK`	space tab
`CNTRL`	`!PRINT`
`DIGIT`	0-9
`GRAPH`	`ALNUM || PUNCT`
`LOWER`	a-z
`PRINT`	`GRAPH ||` space
`PUNCT`	‘~!@#$%^&*()_-=+[{]}|;:’”,<.>/?
`SPACE`	space tab n r f v
`UPPER`	A-Z
`XDIGIT`	0-9A-Fa-f

Note that, if the host character set is ASCII or a superset thereof, all these macros will return false for all values of `char` outside the range of 7-bit ASCII. In particular, both ISPRINT and ISCNTRL return false for characters with numeric values from 128 to 255.

ISIDNUM(c)

ISIDST(c)

IS_VSPACE(c)

IS_NVSPACE(c)

IS_SPACE_OR_NUL(c)

IS_ISOBASIC(c)

These six macros are defined by safe-ctype.h and provide additional character classes which are useful when doing lexical analysis of C or similar languages. They are true for the following sets of characters:

`IDNUM`	A-Za-z0-9_
`IDST`	A-Za-z_
`VSPACE`	r n
`NVSPACE`	space tab f v 0
`SPACE_OR_NUL`	`VSPACE || NVSPACE`
`ISOBASIC`	`VSPACE || NVSPACE || PRINT`

const char ***lbasename**(const char *name)

Given a pointer to a string containing a typical pathname (`/usr/src/cmd/ls/ls.c` for example), returns a pointer to the last component of the pathname (`ls.c` in this case). The returned pointer is guaranteed to lie within the original string. This latter fact is not true of many vendor C libraries, which return special strings or modify the passed strings for particular input.

In particular, the empty string returns the same empty string, and a path ending in `/` returns the empty string after it.

const char ***lrealpath**(const char *name)

Given a pointer to a string containing a pathname, returns a canonical version of the filename. Symlinks will be resolved, and '.' and '..' components will be simplified. The returned value will be allocated using `malloc`, or `NULL` will be returned on a memory allocation error.

const char ***make_relative_prefix**(const char *progname, const char *bin_prefix, const char *prefix)

Given three paths *progname*, *bin_prefix*, *prefix*, return the path that is in the same position relative to *progname* 's directory as *prefix* is relative to *bin_prefix*. That is, a string starting with the directory portion of *progname*, followed by a relative pathname of the difference between *bin_prefix* and *prefix*.

If *progname* does not contain any directory separators, `make_relative_prefix` will search `PATH` to find a program named *progname*. Also, if *progname* is a symbolic link, the symbolic link will be resolved.

For example, if *bin_prefix* is `/alpha/beta/gamma/gcc/delta`, *prefix* is `/alpha/beta/gamma/omega/`, and *progname* is `/red/green/blue/gcc`, then this function will return `/red/green/blue/../../omega/`.

The return value is normally allocated via `malloc`. If no relative prefix can be found, return `NULL`.

char ***make_temp_file**(const char *suffix)

Return a temporary file name (as a string) or `NULL` if unable to create one. *suffix* is a suffix to append to the file name. The string is `malloc` ed, and the temporary file has been created.

void ***memchr**(const void *s, int c, size_t n)

This function searches memory starting at `*s` for the character *c*. The search only ends with the first occurrence of *c*, or after *length* characters; in particular, a null character does not terminate the search. If the character *c* is found within *length* characters of `*s`, a pointer to the character is returned. If *c* is not found, then `NULL` is returned.

int **memcmp**(const void *x, const void *y, size_t count)

Compares the first *count* bytes of two areas of memory. Returns zero if they are the same, a value less than zero if *x* is lexically less than *y*, or a value greater than zero if *x* is lexically greater than *y*. Note that lexical order is determined as if comparing unsigned char arrays.

void ***memcpy**(void *out, const void *in, size_t length)

Copies *length* bytes from memory region *in* to region *out*. Returns a pointer to *out*.

void ***memmem**(const void *haystack, size_t haystack_len, const void *needle, size_t needle_len)

Returns a pointer to the first occurrence of *needle* (length *needle_len*) in *haystack* (length *haystack_len*). Returns `NULL` if not found.

void ***memmove**(void *from, const void *to, size_t count)

Copies *count* bytes from memory area *from* to memory area *to*, returning a pointer to *to*.

void ***mempcpy**(void *out, const void *in, size_t length)

Copies *length* bytes from memory region *in* to region *out*. Returns a pointer to *out* + *length*.

void ***memset**(void *s, int c, size_t count)

Sets the first *count* bytes of *s* to the constant byte *c*, returning a pointer to *s*.

int **mkstemps**(char *pattern, int suffix_len)

Generate a unique temporary file name from *pattern*. *pattern* has the form:

```
path/ccXXXXXXsuffix
```

suffix_len tells us how long *suffix* is (it can be zero length). The last six characters of *pattern* before *suffix* must be `XXXXXX`; they are replaced with a string that makes the filename unique. Returns a file descriptor open on the file for reading and writing.

void **pex_free**(struct pex_obj obj)

Clean up and free all data associated with *obj*. If you have not yet called `pex_get_times` or `pex_get_status`, this will try to kill the subprocesses.

int **pex_get_status**(struct pex_obj *obj, int count, int *vector)

Returns the exit status of all programs run using *obj*. *count* is the number of results expected. The results will be placed into *vector*. The results are in the order of the calls to `pex_run`. Returns 0 on error, 1 on success.

int **pex_get_times**(struct pex_obj *obj, int count, struct pex_time *vector)

Returns the process execution times of all programs run using *obj*. *count* is the number of results expected. The results will be placed into *vector*. The results are in the order of the calls to `pex_run`. Returns 0 on error, 1 on success.

`struct pex_time` has the following fields of the type `unsigned long` : `user_seconds`, `user_microseconds`, `system_seconds`, `system_microseconds`. On systems which do not support reporting process times, all the fields will be set to `0`.

struct pex_obj ***pex_init**(int flags, const char *pname, const char *tempbase)

Prepare to execute one or more programs, with standard output of each program fed to standard input of the next. This is a system independent interface to execute a pipeline.

flags is a bitwise combination of the following:

PEX_RECORD_TIMES

Record subprocess times if possible.

PEX_USE_PIPES

Use pipes for communication between processes, if possible.

PEX_SAVE_TEMPS

Don't delete temporary files used for communication between processes.

pname is the name of program to be executed, used in error messages. *tempbase* is a base name to use for any required temporary files; it may be NULL to use a randomly chosen name.

FILE ***pex_input_file**(struct pex_obj *obj, int flags, const char *in_name)

Return a stream for a temporary file to pass to the first program in the pipeline as input.

The name of the input file is chosen according to the same rules pex_run uses to choose output file names, based on *in_name*, *obj* and the PEX_SUFFIX bit in *flags*.

Don't call fclose on the returned stream; the first call to pex_run closes it automatically.

If *flags* includes PEX_BINARY_OUTPUT, open the stream in binary mode; otherwise, open it in the default mode. Including PEX_BINARY_OUTPUT in *flags* has no effect on Unix.

FILE ***pex_input_pipe**(struct pex_obj *obj, int binary)

Return a stream *fp* for a pipe connected to the standard input of the first program in the pipeline; *fp* is opened for writing. You must have passed PEX_USE_PIPES to the pex_init call that returned *obj*.

You must close *fp* using fclose yourself when you have finished writing data to the pipeline.

The file descriptor underlying *fp* is marked not to be inherited by child processes.

On systems that do not support pipes, this function returns NULL, and sets errno to EINVAL. If you would like to write code that is portable to all systems the pex functions support, consider using pex_input_file instead.

There are two opportunities for deadlock using pex_input_pipe :

- Most systems' pipes can buffer only a fixed amount of data; a process that writes to a full pipe blocks. Thus, if you write to fp before starting the first process, you run the risk of blocking when there is no child process yet to read the data and allow you to continue. pex_input_pipe makes no promises about the size of the pipe's buffer, so if you need to write any data at all before starting the first process in the pipeline, consider using pex_input_file instead.
- Using pex_input_pipe and pex_read_output together may also cause deadlock. If the output pipe fills up, so that each program in the pipeline is waiting for the next to read more data, and you fill the input pipe by writing more data to *fp*, then there is no way to make progress: the only process that could read data from the output pipe is you, but you are blocked on the input pipe.

const char ***pex_one**(int flags, const char *executable, char *const *argv, const char *pname, const char *outname, const char *errname, int *status, int *err)

An interface to permit the easy execution of a single program. The return value and most of the parameters are as for a call to pex_run. *flags* is restricted to a combination of PEX_SEARCH, PEX_STDERR_TO_STDOUT, and PEX_BINARY_OUTPUT. *outname* is interpreted as if PEX_LAST were set. On a successful return, *status will be set to the exit status of the program.

FILE ***pex_read_err**(struct pex_obj *obj, int binary)

Returns a `FILE` pointer which may be used to read the standard error of the last program in the pipeline. When this is used, `PEX_LAST` should not be used in a call to `pex_run`. After this is called, `pex_run` may no longer be called with the same *obj*. *binary* should be non-zero if the file should be opened in binary mode. Don't call `fclose` on the returned file; it will be closed by `pex_free`.

FILE ***pex_read_output**(struct pex_obj *obj, int binary)

Returns a `FILE` pointer which may be used to read the standard output of the last program in the pipeline. When this is used, `PEX_LAST` should not be used in a call to `pex_run`. After this is called, `pex_run` may no longer be called with the same *obj*. *binary* should be non-zero if the file should be opened in binary mode. Don't call `fclose` on the returned file; it will be closed by `pex_free`.

const char ***pex_run**(struct pex_obj *obj, int flags, const char *executable, char *const *argv, const char *outname, const char *errname, int *err)

Execute one program in a pipeline. On success this returns `NULL`. On failure it returns an error message, a statically allocated string.

obj is returned by a previous call to `pex_init`.

flags is a bitwise combination of the following:

PEX_LAST

This must be set on the last program in the pipeline. In particular, it should be set when executing a single program. The standard output of the program will be sent to *outname*, or, if *outname* is `NULL`, to the standard output of the calling program. Do *not* set this bit if you want to call `pex_read_output` (described below). After a call to `pex_run` with this bit set, *pex_run* may no longer be called with the same *obj*.

PEX_SEARCH

Search for the program using the user's executable search path.

PEX_SUFFIX

outname is a suffix. See the description of *outname*, below.

PEX_STDERR_TO_STDOUT

Send the program's standard error to standard output, if possible.

PEX_BINARY_INPUT

The standard input (output or error) of the program should be read (written) in binary mode rather than text mode. These flags are ignored on systems which do not distinguish binary mode and text mode, such as Unix. For proper behavior these flags should match appropriately—a call to `pex_run` using `PEX_BINARY_OUTPUT` should be followed by a call using `PEX_BINARY_INPUT`.

PEX_STDERR_TO_PIPE

Send the program's standard error to a pipe, if possible. This flag cannot be specified together with `PEX_STDERR_TO_STDOUT`. This flag can be specified only on the last program in pipeline.

executable is the program to execute. *argv* is the set of arguments to pass to the program; normally `argv[0]` will be a copy of *executable*.

outname is used to set the name of the file to use for standard output. There are two cases in which no output file will be used:

- if `PEX_LAST` is not set in *flags*, and `PEX_USE_PIPES` was set in the call to `pex_init`, and the system supports pipes
- if `PEX_LAST` is set in *flags*, and *outname* is `NULL`

Otherwise the code will use a file to hold standard output. If `PEX_LAST` is not set, this file is considered to be a temporary file, and it will be removed when no longer needed, unless `PEX_SAVE_TEMPS` was set in the call to `pex_init`.

There are two cases to consider when setting the name of the file to hold standard output.

- `PEX_SUFFIX` is set in *flags*. In this case *outname* may not be `NULL`. If the *tempbase* parameter to `pex_init` was not `NULL`, then the output file name is the concatenation of *tempbase* and *outname*. If *tempbase* was `NULL`, then the output file name is a random file name ending in *outname*.
- `PEX_SUFFIX` was not set in *flags*. In this case, if *outname* is not `NULL`, it is used as the output file name. If *outname* is `NULL`, and *tempbase* was not NULL, the output file name is randomly chosen using *tempbase*. Otherwise the output file name is chosen completely at random.

errname is the file name to use for standard error output. If it is `NULL`, standard error is the same as the caller's. Otherwise, standard error is written to the named file.

On an error return, the code sets `*err` to an `errno` value, or to 0 if there is no relevant `errno`.

const char ***pex_run_in_environment**(struct pex_obj *obj, int flags, const char *executable, char *const *argv, char *const *env, int env_size, const char *outname, const char *errname, int *err)

Execute one program in a pipeline, permitting the environment for the program to be specified. Behaviour and parameters not listed below are as for `pex_run`.

env is the environment for the child process, specified as an array of character pointers. Each element of the array should point to a string of the form `VAR=VALUE`, with the exception of the last element that must be `NULL`.

int **pexecute**(const char *program, char *const *argv, const char *this_pname, const char *temp_base, char **errmsg_fmt, char **errmsg_arg, int flags)

This is the old interface to execute one or more programs. It is still supported for compatibility purposes, but is no longer documented.

void **psignal**(int signo, char *message)

Print *message* to the standard error, followed by a colon, followed by the description of the signal specified by *signo*, followed by a newline.

int **putenv**(const char *string)

Uses `setenv` or `unsetenv` to put *string* into the environment or remove it. If *string* is of the form `name=value` the string is added; if no `=` is present the name is unset/removed.

int **pwait**(int pid, int *status, int flags)

Another part of the old execution interface.

long int **random**(void)

void **srandom**(unsigned int seed)

void ***initstate**(unsigned int seed, void *arg_state, unsigned long n)

void ***setstate**(void *arg_state)

Random number functions. `random` returns a random number in the range 0 to `LONG_MAX`. `srandom` initializes the random number generator to some starting point determined by *seed* (else, the values returned by `random` are always the same for each run of the program). `initstate` and `setstate` allow fine-grained control over the state of the random number generator.

char* reconcat (char *optr, const char *s1, ..., NULL)

Same as `concat`, except that if *optr* is not `NULL` it is freed after the string is created. This is intended to be useful when you're extending an existing string or building up a string in a loop:

```
str = reconcat (str, "pre-", str, NULL);
```

int **rename**(const char *old, const char *new)

Renames a file from *old* to *new*. If *new* already exists, it is removed.

char ***rindex**(const char *s, int c)

Returns a pointer to the last occurrence of the character *c* in the string *s*, or `NULL` if not found. The use of `rindex` is deprecated in new programs in favor of `strrchr`.

int **setenv**(const char *name, const char *value, int overwrite)

void **unsetenv**(const char *name)

`setenv` adds *name* to the environment with value *value*. If the name was already present in the environment, the new value will be stored only if *overwrite* is nonzero. The companion `unsetenv` function removes *name* from the environment. This implementation is not safe for multithreaded code.

void **setproctitle**(const char *fmt, ...)

Set the title of a process to *fmt*. va args not supported for now, but defined for compatibility with BSD.

int **signo_max**(void)

Returns the maximum signal value for which a corresponding symbolic name or message is available. Note that in the case where we use the `sys_siglist` supplied by the system, it is possible for there to be more symbolic names than messages, or vice versa. In fact, the manual page for `psignal(3b)` explicitly warns that one should check the size of the table (`NSIG`) before indexing it, since new signal codes may be added to the system before they are added to the table. Thus `NSIG` might be smaller than value implied by the largest signo value defined in `<signal.h>`.

We return the maximum value that can be used to obtain a meaningful symbolic name or message.

int **sigsetmask**(int set)

Sets the signal mask to the one provided in *set* and returns the old mask (which, for libiberty's implementation, will always be the value 1).

const char ***simple_object_attributes_compare**(simple_object_attributes *attrs1, simple_object_attributes *attrs2, int *err)

Compare *attrs1* and *attrs2*. If they could be linked together without error, return NULL. Otherwise, return an error message and set *err to an errno value or 0 if there is no relevant errno.

simple_object_attributes ***simple_object_fetch_attributes**(simple_object_read *simple_object, const char **errmsg, int *err)

Fetch the attributes of *simple_object*. The attributes are internal information such as the format of the object file, or the architecture it was compiled for. This information will persist until simple_object_attributes_release is called, even if *simple_object* itself is released.

On error this returns NULL, sets *errmsg to an error message, and sets *err to an errno value or 0 if there is no relevant errno.

int **simple_object_find_section**(simple_object_read *simple_object, off_t *offset, off_t *length, const char **errmsg, int *err)

Look for the section *name* in *simple_object*. This returns information for the first section with that name.

If found, return 1 and set *offset to the offset in the file of the section contents and set *length to the length of the section contents. The value in *offset will be relative to the offset passed to simple_object_open_read.

If the section is not found, and no error occurs, simple_object_find_section returns 0 and set *errmsg to NULL.

If an error occurs, simple_object_find_section returns 0, sets *errmsg to an error message, and sets *err to an errno value or 0 if there is no relevant errno.

const char ***simple_object_find_sections**(simple_object_read *simple_object, int (*pfn)(void *data, const char *name, off_t offset, off_t length), void *data, int *err)

This function calls *pfn* for each section in *simple_object*. It calls *pfn* with the section name, the offset within the file of the section contents, and the length of the section contents. The offset within the file is relative to the offset passed to simple_object_open_read. The *data* argument to this function is passed along to *pfn*.

If *pfn* returns 0, the loop over the sections stops and simple_object_find_sections returns. If *pfn* returns some other value, the loop continues.

On success simple_object_find_sections returns. On error it returns an error string, and sets *err to an errno value or 0 if there is no relevant errno.

simple_object_read ***simple_object_open_read**(int descriptor, off_t offset, const char *segment_name, const char **errmsg, int *err)

Opens an object file for reading. Creates and returns an `simple_object_read` pointer which may be passed to other functions to extract data from the object file.

descriptor holds a file descriptor which permits reading.

offset is the offset into the file; this will be `0` in the normal case, but may be a different value when reading an object file in an archive file.

segment_name is only used with the Mach-O file format used on Darwin aka Mac OS X. It is required on that platform, and means to only look at sections within the segment with that name. The parameter is ignored on other systems.

If an error occurs, this functions returns `NULL` and sets `*errmsg` to an error string and sets `*err` to an errno value or `0` if there is no relevant errno.

void **simple_object_release_attributes**(simple_object_attributes *attrs)

Release all resources associated with *attrs*.

void **simple_object_release_read**(simple_object_read *simple_object)

Release all resources associated with *simple_object*. This does not close the file descriptor.

void **simple_object_release_write**(simple_object_write *simple_object)

Release all resources associated with *simple_object*.

simple_object_write ***simple_object_start_write**(simple_object_attributes attrs, const char *segment_name, const char **errmsg, int *err)

Start creating a new object file using the object file format described in *attrs*. You must fetch attribute information from an existing object file before you can create a new one. There is currently no support for creating an object file de novo.

segment_name is only used with Mach-O as found on Darwin aka Mac OS X. The parameter is required on that target. It means that all sections are created within the named segment. It is ignored for other object file formats.

On error `simple_object_start_write` returns `NULL`, sets `*ERRMSG` to an error message, and sets `*err` to an errno value or `0` if there is no relevant errno.

const char ***simple_object_write_add_data**(simple_object_write *simple_object, simple_object_write_section *section, const void *buffer, size_t size, int copy, int *err)

Add data *buffer* / *size* to *section* in *simple_object*. If *copy* is non-zero, the data will be copied into memory if necessary. If *copy* is zero, *buffer* must persist until `simple_object_write_to_file` is called. is released.

On success this returns `NULL`. On error this returns an error message, and sets `*err` to an errno value or 0 if there is no relevant erro.

simple_object_write_section ***simple_object_write_create_section**(simple_object_write *simple_object, const char *name, unsigned int align, const char **errmsg, int *err)

Add a section to *simple_object*. *name* is the name of the new section. *align* is the required alignment expressed as the number of required low-order 0 bits (e.g., 2 for alignment to a 32-bit boundary).

The section is created as containing data, readable, not writable, not executable, not loaded at runtime. The section is not written to the file until `simple_object_write_to_file` is called.

On error this returns `NULL`, sets `*errmsg` to an error message, and sets `*err` to an errno value or `0` if there is no relevant errno.

const char ***simple_object_write_to_file**(simple_object_write *simple_object, int descriptor, int *err)

Write the complete object file to *descriptor*, an open file descriptor. This writes out all the data accumulated by calls to `simple_object_write_create_section` and *simple_object_write_add_data*.

This returns `NULL` on success. On error this returns an error message and sets `*err` to an errno value or `0` if there is no relevant errno.

int **snprintf**(char *buf, size_t n, const char *format, ...)

This function is similar to `sprintf`, but it will write to *buf* at most `n-1` bytes of text, followed by a terminating null byte, for a total of *n* bytes. On error the return value is -1, otherwise it returns the number of bytes, not including the terminating null byte, that would have been written had *n* been sufficiently large, regardless of the actual value of *n*. Note some pre-C99 system libraries do not implement this correctly so users cannot generally rely on the return value if the system version of this function is used.

char ***spaces**(int count)

Returns a pointer to a memory region filled with the specified number of spaces and null terminated. The returned pointer is valid until at least the next call.

splay_tree **splay_tree_new_with_typed_alloc**(splay_tree_compare_fn compare_fn, splay_tree_delete_key_fn delete_key_fn, splay_tree_delete_value_fn delete_value_fn, splay_tree_allocate_fn tree_allocate_fn, splay_tree_allocate_fn node_allocate_fn, splay_tree_deallocate_fn deallocate_fn, void *allocate_data)

This function creates a splay tree that uses two different allocators *tree_allocate_fn* and *node_allocate_fn* to use for allocating the tree itself and its nodes respectively. This is useful when variables of different types need to be allocated with different allocators.

The splay tree will use *compare_fn* to compare nodes, *delete_key_fn* to deallocate keys, and *delete_value_fn* to deallocate values. Keys and values will be deallocated when the tree is deleted using splay_tree_delete or when a node is removed using splay_tree_remove. splay_tree_insert will release the previously inserted key and value using *delete_key_fn* and *delete_value_fn* if the inserted key is already found in the tree.

void **stack_limit_increase**(unsigned long pref)

Attempt to increase stack size limit to *pref* bytes if possible.

char ***stpcpy**(char *dst, const char *src)

Copies the string *src* into *dst*. Returns a pointer to *dst* + strlen(*src*).

char ***stpncpy**(char *dst, const char *src, size_t len)

Copies the string *src* into *dst*, copying exactly *len* and padding with zeros if necessary. If *len* < strlen(*src*) then return *dst* + *len*, otherwise returns *dst* + strlen(*src*).

int **strcasecmp**(const char *s1, const char *s2)

A case-insensitive `strcmp`.

char ***strchr**(const char *s, int c)

Returns a pointer to the first occurrence of the character *c* in the string *s*, or `NULL` if not found. If *c* is itself the null character, the results are undefined.

char ***strdup**(const char *s)

Returns a pointer to a copy of *s* in memory obtained from `malloc`, or `NULL` if insufficient memory was available.

const char ***strerrno**(int errnum)

Given an error number returned from a system call (typically returned in `errno`), returns a pointer to a string containing the symbolic name of that error number, as found in `<errno.h>`.

If the supplied error number is within the valid range of indices for symbolic names, but no name is available for the particular error number, then returns the string `Error` *num*, where *num* is the error number.

If the supplied error number is not within the range of valid indices, then returns `NULL`.

The contents of the location pointed to are only guaranteed to be valid until the next call to `strerrno`.

char ***strerror**(int errnoval)

Maps an `errno` number to an error message string, the contents of which are implementation defined. On systems which have the external variables `sys_nerr` and `sys_errlist`, these strings will be the same as the ones used by `perror`.

If the supplied error number is within the valid range of indices for the `sys_errlist`, but no message is available for the particular error number, then returns the string `Error` *num*, where *num* is the error number.

If the supplied error number is not a valid index into `sys_errlist`, returns `NULL`.

The returned string is only guaranteed to be valid only until the next call to `strerror`.

int **strncasecmp**(const char *s1, const char *s2)

A case-insensitive `strncmp`.

int **strncmp**(const char *s1, const char *s2, size_t n)

Compares the first *n* bytes of two strings, returning a value as `strcmp`.

char ***strndup**(const char *s, size_t n)

Returns a pointer to a copy of *s* with at most *n* characters in memory obtained from `malloc`, or `NULL` if insufficient memory was available. The result is always NUL terminated.

size_t **strnlen**(const char *s, size_t maxlen)

Returns the length of *s*, as with `strlen`, but never looks past the first *maxlen* characters in the string. If there is no '0' character in the first *maxlen* characters, returns *maxlen*.

char ***strrchr**(const char *s, int c)

Returns a pointer to the last occurrence of the character *c* in the string *s*, or `NULL` if not found. If *c* is itself the null character, the results are undefined.

const char ***strsignal**(int signo)

Maps an signal number to an signal message string, the contents of which are implementation defined. On systems which have the external variable `sys_siglist`, these strings will be the same as the ones used by `psignal()`.

If the supplied signal number is within the valid range of indices for the `sys_siglist`, but no message is available for the particular signal number, then returns the string `Signal` *num*, where *num* is the signal number.

If the supplied signal number is not a valid index into `sys_siglist`, returns `NULL`.

The returned string is only guaranteed to be valid only until the next call to `strsignal`.

const char ***strsigno**(int signo)

Given an signal number, returns a pointer to a string containing the symbolic name of that signal number, as found in `<signal.h>`.

If the supplied signal number is within the valid range of indices for symbolic names, but no name is available for the particular signal number, then returns the string `Signal` *num*, where *num* is the signal number.

If the supplied signal number is not within the range of valid indices, then returns `NULL`.

The contents of the location pointed to are only guaranteed to be valid until the next call to `strsigno`.

char ***strstr**(const char *string, const char *sub)

This function searches for the substring *sub* in the string *string*, not including the terminating null characters. A pointer to the first occurrence of *sub* is returned, or `NULL` if the substring is absent. If *sub* points to a string with zero length, the function returns *string*.

double **strtod**(const char *string, char **endptr)

This ISO C function converts the initial portion of *string* to a `double`. If *endptr* is not `NULL`, a pointer to the character after the last character used in the conversion is stored in the location referenced by *endptr*. If no conversion is performed, zero is returned and the value of *string* is stored in the location referenced by *endptr*.

int **strtoerrno**(const char *name)

Given the symbolic name of a error number (e.g., `EACCES`), map it to an errno value. If no translation is found, returns 0.

long int **strtol**(const char *string, char **endptr, int base) unsigned long int **strtoul**(const char *string, char **endptr, int base)

The `strtol` function converts the string in *string* to a long integer value according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0. If *base* is 0, `strtol` will look for the prefixes `0` and `0x` to indicate bases 8 and 16, respectively, else default to base 10. When the base is 16 (either explicitly or implicitly), a prefix of `0x` is allowed. The handling of *endptr* is as that of `strtod` above. The `strtoul` function is the same, except that the converted value is unsigned.

long long int **strtoll**(const char *string, char **endptr, int base)

unsigned long long int **strtoull**(const char *string, char **endptr, int base)

The `strtoll` function converts the string in *string* to a long long integer value according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0. If *base* is 0, `strtoll` will look for the prefixes `0` and `0x` to indicate bases 8 and 16, respectively, else default to base 10. When the base is 16 (either explicitly or implicitly), a prefix of `0x` is allowed. The handling of *endptr* is as that of `strtod` above. The `strtoull` function is the same, except that the converted value is unsigned.

int **strtosigno**(const char *name)

Given the symbolic name of a signal, map it to a signal number. If no translation is found, returns 0.

int **strverscmp**(const char *s1, const char *s2)

The `strverscmp` function compares the string *s1* against *s2*, considering them as holding indices/version numbers. Return value follows the same conventions as found in the `strverscmp` function. In fact, if *s1* and *s2* contain no digits, `strverscmp` behaves like `strcmp`.

Basically, we compare strings normally (character by character), until we find a digit in each string - then we enter a special comparison mode, where each sequence of digits is taken as a whole. If we reach the end of these two parts without noticing a difference, we return to the standard comparison mode. There are two types of numeric parts: “integral” and “fractional” (those begin with a ‘0’). The types of the numeric parts affect the way we sort them:

- integral/integral: we compare values as you would expect.
- fractional/integral: the fractional part is less than the integral one. Again, no surprise.
- fractional/fractional: the things become a bit more complex. If the common prefix contains only leading zeroes, the longest part is less than the other one; else the comparison behaves normally.

```
strverscmp ("no digit", "no digit")
    ⇒ 0    // same behavior as strcmp.
strverscmp ("item#99", "item#100")
    ⇒ <0   // same prefix, but 99 < 100.
strverscmp ("alpha1", "alpha001")
    ⇒ >0   // fractional part inferior to integral one.
strverscmp ("part1_f012", "part1_f01")
    ⇒ >0   // two fractional parts.
strverscmp ("foo.009", "foo.0")
    ⇒ <0   // idem, but with leading zeroes only.
```

This function is especially useful when dealing with filename sorting, because filenames frequently hold indices/version numbers.

void **timeval_add**(struct timeval *a, struct timeval *b, struct timeval *result)

Adds *a* to *b* and stores the result in *result*.

void **timeval_sub**(struct timeval *a, struct timeval *b, struct timeval *result)

Subtracts *b* from *a* and stores the result in *result*.

char ***tmpnam**(char *s)

This function attempts to create a name for a temporary file, which will be a valid file name yet not exist when `tmpnam` checks for it. *s* must point to a buffer of at least `L_tmpnam` bytes, or be `NULL`. Use of this function creates a security risk, and it must not be used in new projects. Use `mkstemp` instead.

int **unlink_if_ordinary**(const char*)

Unlinks the named file, unless it is special (e.g. a device file). Returns 0 when the file was unlinked, a negative value (and errno set) when there was an error deleting the file, and a positive value if no attempt was made to unlink the file because it is special.

void **unlock_std_streams**(void)

If the OS supports it, ensure that the standard I/O streams, `stdin`, `stdout` and `stderr` are setup to avoid any multi-threaded locking. Otherwise do nothing.

void **unlock_stream**(FILE *stream)

If the OS supports it, ensure that the supplied stream is setup to avoid any multi-threaded locking. Otherwise leave the `FILE` pointer unchanged. If the *stream* is `NULL` do nothing.

int **vasprintf**(char **resptr, const char *format, va_list args)

Like `vsprintf`, but instead of passing a pointer to a buffer, you pass a pointer to a pointer. This function will compute the size of the buffer needed, allocate memory with `malloc`, and store a pointer to the allocated memory in `*resptr`. The value returned is the same as `vsprintf` would return. If memory could not be allocated, minus one is returned and `NULL` is stored in `*resptr`.

int **vfork**(void)

Emulates `vfork` by calling `fork` and returning its value.

int **vprintf**(const char *format, va_list ap)

int **vfprintf**(FILE *stream, const char *format, va_list ap)

int **vsprintf**(char *str, const char *format, va_list ap)

These functions are the same as `printf`, `fprintf`, and `sprintf`, respectively, except that they are called with a `va_list` instead of a variable number of arguments. Note that they do not call `va_end` ; this is the application's responsibility. In `libiberty` they are implemented in terms of the nonstandard but common function `_doprnt`.

int **vsnprintf**(char *buf, size_t n, const char *format, va_list ap)

This function is similar to `vsprintf`, but it will write to *buf* at most `n-1` bytes of text, followed by a terminating null byte, for a total of *n* bytes. On error the return value is -1, otherwise it returns the number of characters that would have been printed had *n* been sufficiently large, regardless of the actual value of *n*. Note some pre-C99 system libraries do not implement this correctly so users cannot generally rely on the return value if the system version of this function is used.

int **waitpid**(int pid, int *status, int)

This is a wrapper around the `wait` function. Any 'special' values of *pid* depend on your implementation of `wait`, as does the return value. The third argument is unused in `libiberty`.

int **writeargv**(char *const *argv, FILE *file)

Write each member of ARGV, handling all necessary quoting, to the file named by FILE, separated by whitespace. Return 0 on success, non-zero if an error occurred while writing to FILE.

char ***xasprintf**(const char *format, ...)

Print to allocated string without fail. If `xasprintf` fails, this will print a message to `stderr` (using the name set by `xmalloc_set_program_name`, if any) and then call `xexit`.

int **xatexit**(void (*fn)(void))

Behaves as the standard `atexit` function, but with no limit on the number of registered functions. Returns 0 on success, or -1 on failure. If you use `xatexit` to register functions, you must use `xexit` to terminate your program.

void ***xcalloc**(size_t nelem, size_t elsize)

Allocate memory without fail, and set it to zero. This routine functions like `calloc`, but will behave the same as `xmalloc` if memory cannot be found.

void **xexit**(int code)

Terminates the program. If any functions have been registered with the `xatexit` replacement function, they will be called first. Termination is handled via the system's normal `exit` call.

void ***xmalloc**(size_t)

Allocate memory without fail. If `malloc` fails, this will print a message to `stderr` (using the name set by `xmalloc_set_program_name`, if any) and then call `xexit`. Note that it is therefore safe for a program to contain `#define malloc xmalloc` in its source.

void **xmalloc_failed**(size_t)

This function is not meant to be called by client code, and is listed here for completeness only. If any of the allocation routines fail, this function will be called to print an error message and terminate execution.

void **xmalloc_set_program_name**(const char *name)

You can use this to set the name of the program used by `xmalloc_failed` when printing a failure message.

void ***xmemdup**(void *input, size_t copy_size, size_t alloc_size)

Duplicates a region of memory without fail. First, *alloc_size* bytes are allocated, then *copy_size* bytes from *input* are copied into it, and the new memory is returned. If fewer bytes are copied than were allocated, the remaining memory is zeroed.

void *xrealloc(void *ptr, size_t size)

Reallocate memory without fail. This routine functions like realloc, but will behave the same as xmalloc if memory cannot be found.

char *xstrdup(const char *s)

Duplicates a character string without fail, using xmalloc to obtain memory.

char *xstrerror(int errnum)

Behaves exactly like the standard strerror function, but will never return a NULL pointer.

char *xstrndup(const char *s, size_t n)

Returns a pointer to a copy of *s* with at most *n* characters without fail, using xmalloc to obtain memory. The result is always NUL terminated.

char *xvasprintf(const char *format, va_list args)

Print to allocated string without fail. If xvasprintf fails, this will print a message to stderr (using the name set by xmalloc_set_program_name, if any) and then call xexit.

CHAPTER

SIX

GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

6.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the *Lesser* General Public License because it does *Less* to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a 'work based on the library' and a 'work that uses the library'. The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

1. **This License Agreement applies to any software library or other program**
   which contains a notice placed by the copyright holder or other authorized party saying

it may be distributed under the terms of this Lesser General Public License (also called 'this License'). Each licensee is addressed as 'you'.

A 'library' means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The 'Library', below, refers to any such software library or work which has been distributed under these terms. A 'work based on the Library' means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term 'modification'.)

'Source code' for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

2. **You may copy and distribute verbatim copies of the Library's**
   complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. **You may modify your copy or copies of the Library or any portion**
   of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a* The modified work must itself be a software library.

   - You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
   - You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
   - If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. **You may opt to apply the terms of the ordinary GNU General Public**
   License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

5. **You may copy and distribute the Library (or a portion or**
   derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

6. **A program that contains no derivative of any portion of the**
   Library, but is designed to work with the Library by being compiled or linked with it, is called a 'work that uses the Library'. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a 'work that uses the Library' with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a 'work that uses the library'. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a 'work that uses the Library' uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

7. **As an exception to the Sections above, you may also combine or**
   link a 'work that uses the Library' with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   a* Accompany the work with the complete corresponding
   machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable 'work that uses the Library', as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

   - Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
   - Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

- If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the 'work that uses the Library' must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

8. **You may place library facilities that are a work based on the**
   Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a* Accompany the combined library with a copy of the same work
   based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   - Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

9. **You may not copy, modify, sublicense, link with, or distribute**
   the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. **You are not required to accept this License, since you have not**
    signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

11. **Each time you redistribute the Library (or any work based on the**
    Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted

herein. You are not responsible for enforcing compliance by third parties with this License.

12. **If, as a consequence of a court judgment or allegation of patent**
    infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

13. **If the distribution and/or use of the Library is restricted in**
    certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

14. **The Free Software Foundation may publish revised and/or new**
    versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and 'any later version', you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

15. **If you wish to incorporate parts of the Library into other free**
    programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our

decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

16. **BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO**
    WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY 'AS IS' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

17. **IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN**
    WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

6.2 How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the 'copyright' line and a pointer to where the full notice is found.

```
one line to give the library's name and an idea of what it does.
Copyright (C) year  name of author

This library is free software; you can redistribute it and/or modify it
under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 2.1 of the License, or (at
your option) any later version.
```

(continues on next page)

(continued from previous page)

```
This library is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
USA.
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a ‘copyright disclaimer’ for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the library
`Frob' (a library for tweaking knobs) written by James Random Hacker.

signature of Ty Coon, 1 April 1990
Ty Coon, President of Vice
```

That’s all there is to it!

CHAPTER

SEVEN

BSD

Copyright (C) 1990 Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. [rescinded 22 July 1999]
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS 'AS IS' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

INDEX

P

R

S