Chapter 3

User-Level Memory Management

In this chapter

- 3.1 Linux/Unix Address Space page 52
- 3.2 Memory Allocation page 56
- 3.3 Summary page 80
- Exercises page 81

Without memory for storing data, it's impossible for a program to get any work done. (Or rather, it's impossible to get any *useful* work done.) Real-world programs can't afford to rely on fixed-size buffers or arrays of data structures. They have to be able to handle inputs of varying sizes, from small to large. This in turn leads to the use of *dynamically allocated memory*—memory allocated at runtime instead of at compile time. This is how the GNU "no arbitrary limits" principle is put into action.

Because dynamically allocated memory is such a basic building block for real-world programs, we cover it early, before looking at everything else there is to do. Our discussion focuses exclusively on the user-level view of the process and its memory; it has nothing to do with CPU architecture.

3.1 Linux/Unix Address Space

For a working definition, we've said that a *process* is a running program. This means that the operating system has loaded the executable file for the program into memory, has arranged for it to have access to its command-line arguments and environment variables, and has started it running. A process has five conceptually different areas of memory allocated to it:

Code

Often referred to as the *text segment*, this is the area in which the executable instructions reside. Linux and Unix arrange things so that multiple running instances of the same program share their code if possible; only one copy of the instructions for the same program resides in memory at any time. (This is transparent to the running programs.) The portion of the executable file containing the text segment is the *text section*.

Initialized data

Statically allocated and global data that are initialized with nonzero values live in the *data segment*. Each process running the same program has its own data segment. The portion of the executable file containing the data segment is the *data section*.

Zero-initialized data

Global and statically allocated data that are initialized to zero by default are kept in what is colloquially called the *BSS* area of the process.[1] Each process running the same program has its own BSS area. When running, the BSS data are placed in the data segment. In the executable file, they are stored in the *BSS section*.

The format of a Linux/Unix executable is such that only variables that are initialized to a nonzero value occupy space in the executable's disk file. Thus, a large array declared '`static char somebuf[2048];`', which is automatically zero-filled, does not take up 2 KB worth of disk space. (Some compilers have options that let you place zero-initialized data into the data segment.)

Heap

The *heap* is where dynamic memory (obtained by `malloc()` and friends) comes from. As memory is allocated on the heap, the process's address space grows, as you can see by watching a running program with the `ps` command.

Although it is possible to give memory back to the system and shrink a process's address space, this is almost never done. (We distinguish between releasing no-longer-needed dynamic memory and shrinking the address space; this is discussed in more detail later in this chapter.)

It is typical for the heap to "grow upward." This means that successive items that are added to the heap are added at addresses that are numerically greater than previous items. It is also typical for the heap to start immediately after the BSS area of the data segment.

Stack

The *stack segment* is where local variables are allocated. Local variables are all variables declared inside the opening left brace of a function body (or other left brace) that aren't defined as `static`.

On most architectures, function parameters are also placed on the stack, as well as "invisible" bookkeeping information generated by the compiler, such as room for a function return value and storage for the return address representing the return from a function to its caller. (Some architectures do all this with registers.)

---

[1] BSS is an acronym for "Block Started by Symbol," a mnemonic from the IBM 7094 assembler.

It is the use of a stack for function parameters and return values that makes it convenient to write *recursive* functions (functions that call themselves).

Variables stored on the stack "disappear" when the function containing them returns; the space on the stack is reused for subsequent function calls.

On most modern architectures, the stack "grows downward," meaning that items deeper in the call chain are at numerically lower addresses.

When a program is running, the initialized data, BSS, and heap areas are usually placed into a single contiguous area: the data segment. The stack segment and code segment are separate from the data segment and from each other. This is illustrated in Figure 3.1.

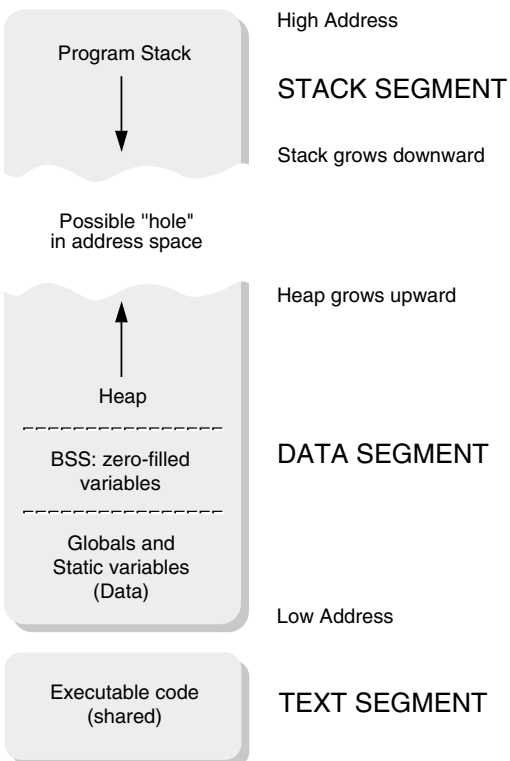

FIGURE 3.1
Linux/Unix process address space

Although it's theoretically possible for the stack and heap to grow into each other, the operating system prevents that event, and any program that tries to make it happen is asking for trouble. This is particularly true on modern systems, on which process address spaces are large and the gap between the top of the stack and the end of the heap is a big one. The different memory areas can have different hardware memory protection assigned to them. For example, the text segment might be marked "execute only," whereas the data and stack segments would have execute permission disabled. This practice can prevent certain kinds of security attacks. The details, of course, are hardware and operating-system specific and likely to change over time. Of note is that both Standard C and C++ allow `const` items to be placed in read-only memory. The relationship among the different segments is summarized in Table 3.1.

TABLE 3.1
Executable program segments and their locations

Program memory	Address space segment	Executable file section
Code	Text	Text
Initialized data	Data	Data
BSS	Data	BSS
Heap	Data	
Stack	Stack	

The `size` program prints out the size in bytes of each of the text, data, and BSS sections, along with the total size in decimal and hexadecimal. (The `ch03-memaddr.c` program is shown later in this chapter; see Section 3.2.5, "Address Space Examination," page 78.)

```
$ cc -O ch03-memaddr.c -o ch03-memaddr              Compile the program
$ ls -l ch03-memaddr                                Show total size
-rwxr-xr-x    1 arnold   devel       12320 Nov 24 16:45 ch03-memaddr
$ size ch03-memaddr                                 Show component sizes
   text    data     bss     dec     hex filename
   1458     276       8    1742     6ce ch03-memaddr
$ strip ch03-memaddr                                Remove symbols
$ ls -l ch03-memaddr                                Show total size again
-rwxr-xr-x    1 arnold   devel        3480 Nov 24 16:45 ch03-memaddr
$ size ch03-memaddr                                 Component sizes haven't changed
   text    data     bss     dec     hex filename
   1458     276       8    1742     6ce ch03-memaddr
```

The total size of what gets loaded into memory is only 1742 bytes, in a file that is 12,320 bytes long. Most of that space is occupied by the *symbols*, a list of the program's variables and function names. (The symbols are not loaded into memory when the program runs.) The `strip` program removes the symbols from the object file. This can save significant disk space for a large program, at the cost of making it impossible to debug a core dump[2] should one occur. (On modern systems this isn't worth the trouble; don't use `strip`.) Even after removing the symbols, the file is still larger than what gets loaded into memory since the object file format maintains additional data about the program, such as what shared libraries it may use, if any.[3]

Finally, we'll mention that *threads* represent multiple threads of execution within a *single* address space. Typically, each thread has its own stack, and a way to get *thread local* data, that is, dynamically allocated data for private use by the thread. We don't otherwise cover threads in this book, since they are an advanced topic.

3.2 Memory Allocation

Four library functions form the basis for dynamic memory management from C. We describe them first, followed by descriptions of the two system calls upon which these library functions are built. The C library functions in turn are usually used to implement other library functions that allocate memory and the C++ `new` and `delete` operators.

Finally, we discuss a function that you will see used frequently, but which we don't recommend.

3.2.1 Library Calls: `malloc()`, `calloc()`, `realloc()`, `free()`

Dynamic memory is allocated by either the `malloc()` or `calloc()` functions. These functions return pointers to the allocated memory. Once you have a block of memory

---

[2] A *core dump* is the memory image of a running process created when the process terminates unexpectedly. It may be used later for debugging. Unix systems named the file `core`, and GNU/Linux systems use `core.`*`pid`*, where *`pid`* is the process ID of the process that died.

[3] The description here is a deliberate simplification. Running programs occupy much more space than the `size` program indicates, since shared libraries are included in the address space. Also, the data segment will grow as a program allocates memory.

of a certain initial size, you can change its size with the `realloc()` function. Dynamic memory is released with the `free()` function.

Debugging the use of dynamic memory is an important topic in its own right. We discuss tools for this purpose in Section 15.5.2, "Memory Allocation Debuggers," page 612.

3.2.1.1 Examining C Language Details

Here are the function declarations from the GNU/Linux *malloc*(3) manpage:

```
#include <stdlib.h>                                          ISO C

void *calloc(size_t nmemb, size_t size);      Allocate and zero fill
void *malloc(size_t size);                    Allocate raw memory
void free(void *ptr);                         Release memory
void *realloc(void *ptr, size_t size);        Change size of existing allocation
```

The allocation functions all return type `void *`. This is a *typeless* or *generic pointer*; all you can do with such a pointer is cast it to a different type and assign it to a typed pointer. Examples are coming up.

The type `size_t` is an unsigned integral type that represents amounts of memory. It is used for dynamic memory allocation, and we see many uses of it throughout the book. On most modern systems, `size_t` is `unsigned long`, but it's better to use `size_t` explicitly than to use a plain `unsigned` integral type.

The `ptrdiff_t` type is used for address calculations in pointer arithmetic, such as calculating where in an array a pointer may be pointing:

```
#define MAXBUF ...
char *p;
char buf[MAXBUF];
ptrdiff_t where;

p = buf;
while (some condition) {
    ...
    p += something ;
    ...
    where = p - buf;    /* what index are we at? */
}
```

The `<stdlib.h>` header file declares many of the standard C library routines and types (such as `size_t`), and it also defines the preprocessor constant `NULL`, which represents the "null" or invalid pointer. (This is a zero value, such as `0` or '`((void *) 0)`'.

The C++ idiom is to use `0` explicitly; in C, however, `NULL` is preferred, and we find it to be much more readable for C code.)

3.2.1.2 Initially Allocating Memory: `malloc()`

Memory is allocated initially with `malloc()`. The value passed in is the total number of bytes requested. The return value is a pointer to the newly allocated memory or `NULL` if memory could not be allocated. In the latter event, `errno` will be set to indicate the error. (`errno` is a special variable that system calls and library functions set to indicate what went wrong. It's described in Section 4.3, "Determining What Went Wrong," page 86.) For example, suppose we wish to allocate a variable number of some structure. The code looks something like this:

```
struct coord {                          /* 3D coordinates */
    int x, y, z;
} *coordinates;
unsigned int count;                      /* how many we need */
size_t amount;                           /* total amount of memory */

/* ... determine count somehow... */
amount = count * sizeof(struct coord);  /* how many bytes to allocate */

coordinates = (struct coord *) malloc(amount);  /* get the space */
if (coordinates == NULL) {
    /* report error, recover or give up */
}
/* ... use coordinates ... */
```

The steps shown here are quite boilerplate. The order is as follows:

1. Declare a pointer of the proper type to point to the allocated memory.
2. Calculate the size *in bytes* of the memory to be allocated. This involves multiplying a count of objects needed by the size of the individual object. This size in turn is retrieved from the C `sizeof` operator, which exists for this purpose (among others). Thus, while the size of a particular `struct` may vary across compilers and architectures, `sizeof` always returns the correct value and the source code remains correct and portable.

   When allocating arrays for character strings or other data of type `char`, it is not necessary to multiply by `sizeof(char)`, since by definition this is always `1`. But it won't hurt anything either.
3. Allocate the storage by calling `malloc()`, assigning the function's return value to the pointer variable. It is good practice to cast the return value of `malloc()`

to that of the variable being assigned to. In C it's not required (although the compiler may generate a warning). We strongly recommend *always* casting the return value.

Note that in C++, assignment of a pointer value of one type to a pointer of another type does requires a cast, whatever the context. For dynamic memory management, C++ programs should use `new` and `delete`, to avoid type problems, and not `malloc()` and `free()`.

4. Check the return value. *Never* assume that memory allocation will succeed. If the allocation fails, `malloc()` returns `NULL`. If you use the value without checking, it is likely that your program will immediately die from a *segmentation violation* (or *segfault*), which is an attempt to use memory not in your address space.

   If you check the return value, you can at least print a diagnostic message and terminate gracefully. Or you can attempt some other method of recovery.

Once we've allocated memory and set `coordinates` to point to it, we can then treat `coordinates` as if it were an array, although it's really a pointer:

```
int cur_x, cur_y, cur_z;
size_t an_index;
an_index = something;
cur_x = coordinates[an_index].x;
cur_y = coordinates[an_index].y;
cur_z = coordinates[an_index].z;
```

The compiler generates correct code for indexing through the pointer to retrieve the members of the structure at `coordinates[an_index]`.

> NOTE The memory returned by `malloc()` is *not* initialized. It can contain any random garbage. You should immediately initialize the memory with valid data or at least with zeros. To do the latter, use `memset()` (discussed in Section 12.2, "Low-Level Memory: The `memXXX()` Functions," page 432):
>
> ```
> memset(coordinates, '\0', amount);
> ```
>
> Another option is to use `calloc()`, described shortly.

Geoff Collyer recommends the following technique for allocating memory:

```
some_type *pointer;

pointer = malloc(count * sizeof(*pointer));
```

This approach guarantees that the `malloc()` will allocate the correct amount of memory without your having to consult the declaration of `pointer`. If `pointer`'s type later changes, the `sizeof` operator automatically ensures that the count of bytes to allocate stays correct. (Geoff's technique omits the cast that we just discussed. Having the cast there also ensures a diagnostic if `pointer`'s type changes and the call to `malloc()` isn't updated.)

3.2.1.3 Releasing Memory: `free()`

When you're done using the memory, you "give it back" by using the `free()` function. The single argument is a pointer previously obtained from one of the other allocation routines. It is safe (although useless) to pass a null pointer to `free()`:

```
free(coordinates);
coordinates = NULL;     /* not required, but a good idea */
```

Once `free(coordinates)` is called, the memory pointed to by `coordinates` is *off limits*. It now "belongs" to the allocation subroutines, and they are free to manage it as they see fit. They can change the contents of the memory or even release it from the process's address space! There are thus several common errors to watch out for with `free()`:

Accessing freed memory

If unchanged, `coordinates` continues to point at memory that no longer belongs to the application. This is called a *dangling pointer*. In many systems, you can get away with continuing to access this memory, at least until the next time more memory is allocated or freed. In many others though, such access won't work.

In sum, accessing freed memory is a bad idea: It's not portable or reliable, and the *GNU Coding Standards* disallows it. For this reason, it's a good idea to immediately set the program's pointer variable to `NULL`. If you then accidentally attempt to access freed memory, your program will immediately fail with a segmentation fault (before you've released it to the world, we hope).

Freeing the same pointer twice

This causes "undefined behavior." Once the memory has been handed back to the allocation routines, they may merge the freed block with other free storage under management. Freeing something that's already been freed is likely to lead to confusion or crashes at best, and so-called double frees have been known to lead to security problems.

Passing a pointer not obtained from `malloc()`, `calloc()`, *or* `realloc()`

This seems obvious, but it's important nonetheless. Even passing in a pointer to somewhere in the middle of dynamically allocated memory is bad:

```
free(coordinates + 10);         /* Release all but first 10 elements. */
```

This call won't work, and it's likely to lead to disastrous consequences, such as a crash. (This is because many `malloc()` implementations keep "bookkeeping" information *in front of* the returned data. When `free()` goes to use that information, it will find invalid data there. Other implementations have the bookkeeping information at the end of the allocated chunk; the same issues apply.)

Buffer overruns and underruns

Accessing memory outside an allocated chunk also leads to undefined behavior, again because this is likely to be bookkeeping information or possibly memory that's not even in the address space. Writing into such memory is much worse, since it's likely to destroy the bookkeeping data.

Failure to free memory

Any dynamic memory that's not needed should be released. In particular, memory that is allocated inside loops or recursive or deeply nested function calls should be carefully managed and released. Failure to take care leads to *memory leaks*, whereby the process's memory can grow without bounds; eventually, the process dies from lack of memory.

This situation can be particularly pernicious if memory is allocated per input record or as some other function of the input: The memory leak won't be noticed when run on small inputs but can suddenly become obvious (and embarrassing) when run on large ones. This error is even worse for systems that must run continuously, such as telephone switching systems. A memory leak that crashes such a system can lead to significant monetary or other damage.

Even if the program never dies for lack of memory, constantly growing programs suffer in performance, because the operating system has to manage keeping in-use data in physical memory. In the worst case, this can lead to behavior known as *thrashing*, whereby the operating system is so busy moving the contents of the address space into and out of physical memory that no real work gets done.

While it's possible for `free()` to hand released memory back to the system and shrink the process address space, this is almost never done. Instead, the released memory is kept available for allocation by the next call to `malloc()`, `calloc()`, or `realloc()`.

Given that released memory continues to reside in the process's address space, it may pay to zero it out before releasing it. Security-sensitive programs may choose to do this, for example.

See Section 15.5.2, "Memory Allocation Debuggers," page 612, for discussion of a number of useful dynamic-memory debugging tools.

3.2.1.4 Changing Size: `realloc()`

Dynamic memory has a significant advantage over statically declared arrays, which is that it's possible to use exactly as much memory as you need, and no more. It's not necessary to declare a global, `static`, or automatic array of some fixed size and hope that it's (a) big enough and (b) not too big. Instead, you can allocate exactly as much as you need, no more and no less.

Additionally, it's possible to change the size of a dynamically allocated memory area. Although it's possible to shrink a block of memory, more typically, the block is grown. Changing the size is handled with `realloc()`. Continuing with the `coordinates` example, typical code goes like this:

```
int new_count;
size_t new_amount;
struct coord *newcoords;

/* set new_count, for example: */
new_count = count * 2;          /* double the storage */
new_amount = new_count * sizeof(struct coord);

newcoords = (struct coord *) realloc(coordinates, new_amount);
if (newcoords == NULL) {
    /* report error, recover or give up */
}

coordinates = newcoords;
/* continue using coordinates ... */
```

As with `malloc()`, the steps are boilerplate in nature and are similar in concept:

1. Compute the new size to allocate, in bytes.
2. Call `realloc()` with the original pointer obtained from `malloc()` (or from `calloc()` or an earlier call to `realloc()`) and the new size.

3. Cast and assign the return value of `realloc()`. More discussion of this shortly.
4. As for `malloc()`, *check* the return value to make sure it's not `NULL`. Any memory allocation routine can fail.

When growing a block of memory, `realloc()` often allocates a new block of the right size, copies the data from the old block into the new one, and returns a pointer to the new one.

When shrinking a block of data, `realloc()` can often just update the internal bookkeeping information and return the same pointer. This saves having to copy the original data. However, if this happens, *don't assume you can still use the memory beyond the new size!*

In either case, you can assume that if `realloc()` doesn't return `NULL`, the old data has been copied for you into the new memory. Furthermore, the old pointer is no longer valid, as if you had called `free()` with it, and you should not use it. This is true of all pointers into that block of data, not just the particular one used to call `free()`.

You may have noticed that our example code used a separate variable to point to the changed storage block. It would be possible (but a bad idea) to use the same initial variable, like so:

```
coordinates = realloc(coordinates, new_amount);
```

This is a bad idea for the following reason. When `realloc()` returns `NULL`, the original pointer is still valid; it's safe to continue using that memory. However, if you reuse the same variable and `realloc()` returns `NULL`, you've now *lost* the pointer to the original memory. That memory can no longer be used. More important, that memory can no longer be freed! This creates a memory leak, which is to be avoided.

There are some special cases for the Standard C version of `realloc()`: When the `ptr` argument is `NULL`, `realloc()` acts like `malloc()` and allocates a fresh block of storage. When the `size` argument is `0`, `realloc()` acts like `free()` and *releases* the memory that `ptr` points to. Because (a) this can be confusing and (b) older systems don't implement this feature, we recommend using `malloc()` when you mean `malloc()` and `free()` when you mean `free()`.

Here is another, fairly subtle, "gotcha."[4] Consider a routine that maintains a `static` pointer to some dynamically allocated data, which the routine occasionally has to grow. It may also maintain automatic (that is, local) pointers into this data. (For brevity, we omit error checking code. In production code, don't do that.) For example:

```
void manage_table(void)
{
    static struct table *table;
    struct table *cur, *p;
    int i;
    size_t count;

    ...
    table = (struct table *) malloc(count * sizeof(struct table));
    /* fill table */
    cur = & table[i];       /* point at i'th item */
    ...
    cur->i = j;             /* use pointer */
    ...
    if (some condition) {    /* need to grow table */
        count += count/2;
        p = (struct table *) realloc(table, count * sizeof(struct table));
        table = p;
    }

    cur->i = j;             /* PROBLEM 1: update table element */

    other_routine();        /* PROBLEM 2: see text */
    cur->j = k;             /* PROBLEM 2: see text */
    ...
}
```

This looks straightforward; `manage_table()` allocates the data, uses it, changes the size, and so on. But there are some problems that don't jump off the page (or the screen) when you are looking at this code.

In the line marked '`PROBLEM 1`', the `cur` pointer is used to update a table element. However, `cur` was assigned on the basis of the *initial* value of `table`. If *`some condition`* was true and `realloc()` returned a different block of memory, `cur` now points into the original, freed memory! Whenever `table` changes, any pointers into the memory need to be updated too. What's missing here is the statement '`cur = & table[i];`' after `table` is reassigned following the call to `realloc()`.

---

[4] It is derived from real-life experience with `gawk`.

The two lines marked 'PROBLEM 2' are even more subtle. In particular, suppose `other_routine()` makes a *recursive* call to `manage_table()`. The `table` variable could be changed again, completely invisibly! Upon return from `other_routine()`, the value of `cur` could once again be invalid.

One might think (as we did) that the only solution is to be aware of this and supply a suitably commented reassignment to `cur` after the function call. However, Brian Kernighan kindly set us straight. If we use indexing, the pointer maintenance issue doesn't even arise:

```
table = (struct table *) malloc(count * sizeof(struct table));
/* fill table */
...
table[i].i = j;         /* Update a member of the i'th element */
...
if (some condition) {   /* need to grow table */
    count += count/2;
    p = (struct table *) realloc(table, count * sizeof(struct table));
    table = p;
}

table[i].i = j;         /* PROBLEM 1 goes away */
other_routine();        /* Recursively calls us, modifies table */
table[i].j = k;         /* PROBLEM 2 goes away also */
```

Using indexing doesn't solve the problem if you have a *global* copy of the original pointer to the allocated data; in that case, you still have to worry about updating your global structures after calling `realloc()`.

> NOTE As with `malloc()`, when you grow a piece of memory, the newly allocated memory returned from `realloc()` is not zero-filled. You must clear it yourself with `memset()` if that's necessary, since `realloc()` only allocates the fresh memory; it doesn't do anything else.

3.2.1.5 Allocating and Zero-filling: `calloc()`

The `calloc()` function is a straightforward wrapper around `malloc()`. Its primary advantage is that it zeros the dynamically allocated memory. It also performs the size calculation for you by taking as parameters the number of items and the size of each:

```
coordinates = (struct coord *) calloc(count, sizeof(struct coord));
```

Conceptually, at least, the `calloc()` code is fairly simple. Here is one possible implementation:

```
void *calloc(size_t nmemb, size_t size)
{
    void *p;
    size_t total;

    total = nmemb * size;                    Compute size
    p = malloc(total);                       Allocate the memory

    if (p != NULL)                           If it worked ...
        memset(p, '\0', total);              Fill it with zeros

    return p;                                Return value is NULL or pointer
}
```

Many experienced programmers prefer to use `calloc()` since then there's never any question about the contents of the newly allocated memory.

Also, if you know you'll need zero-filled memory, you should use `calloc()`, because it's possible that the memory `malloc()` returns is already zero-filled. Although you, the programmer, can't know this, `calloc()` can know about it and avoid the call to `memset()`.

3.2.1.6 Summarizing from the GNU Coding Standards

To summarize, here is what the *GNU Coding Standards* has to say about using the memory allocation routines:

> Check every call to `malloc` or `realloc` to see if it returned zero. Check `realloc` even if you are making the block smaller; in a system that rounds block sizes to a power of 2, `realloc` may get a different block if you ask for less space.
>
> In Unix, `realloc` can destroy the storage block if it returns zero. GNU `realloc` does not have this bug: If it fails, the original block is unchanged. Feel free to assume the bug is fixed. If you wish to run your program on Unix, and wish to avoid lossage in this case, you can use the GNU `malloc`.
>
> You must expect `free` to alter the contents of the block that was freed. Anything you want to fetch from the block, you must fetch before calling `free`.

In three short paragraphs, Richard Stallman has distilled the important principles for doing dynamic memory management with `malloc()`. It is the use of dynamic

memory and the "no arbitrary limits" principle that makes GNU programs so robust and more capable than their Unix counterparts.

We do wish to point out that the C standard requires `realloc()` to *not* destroy the original block if it returns `NULL`.

3.2.1.7 Using Private Allocators

The `malloc()` suite is a general-purpose memory allocator. It has to be able to handle requests for arbitrarily large or small amounts of memory and do all the bookkeeping when different chunks of allocated memory are released. If your program does considerable dynamic memory allocation, you may thus find that it spends a large proportion of its time in the `malloc()` functions.

One thing you can do is write a *private allocator*—a set of functions or macros that allocates large chunks of memory from `malloc()` and then parcels out small chunks one at a time. This technique is particularly useful if you allocate many individual instances of the same relatively small structure.

For example, GNU `awk` (`gawk`) uses this technique. From the file `awk.h` in the `gawk` distribution (edited slightly to fit the page):

```
#define getnode(n)   if (nextfree) n = nextfree, nextfree = nextfree->nextp;\
                     else n = more_nodes()

#define freenode(n)  ((n)->flags = 0, (n)->exec_count = 0,\
                     (n)->nextp = nextfree, nextfree = (n))
```

The `nextfree` variable points to a linked list of `NODE` structures. The `getnode()` macro pulls the first structure off the list if one is there. Otherwise, it calls `more_nodes()` to allocate a new list of free `NODE`s. The `freenode()` macro releases a `NODE` by putting it at the head of the list.

> NOTE When first writing your application, do it the simple way: use `malloc()` and `free()` directly. *If and only if* profiling your program shows you that it's spending a significant amount of time in the memory-allocation functions should you consider writing a private allocator.

3.2.1.8 Example: Reading Arbitrarily Long Lines

Since this is, after all, *Linux Programming by Example*, it's time for a real-life example. The following code is the `readline()` function from GNU Make 3.80

(`ftp://ftp.gnu.org/gnu/make/make-3.80.tar.gz`). It can be found in the file `read.c`.

Following the "no arbitrary limits" principle, lines in a `Makefile` can be of any length. Thus, this routine's primary job is to read lines of any length and make sure that they fit into the buffer being used.

A secondary job is to deal with continuation lines. As in C, lines that end with a backslash logically continue to the next line. The strategy used is to maintain a buffer. As many lines as will fit in the buffer are kept there, with pointers keeping track of the start of the buffer, the current line, and the next line. Here is the structure:

```
struct ebuffer
  {
    char *buffer;       /* Start of the current line in the buffer.  */
    char *bufnext;      /* Start of the next line in the buffer.  */
    char *bufstart;     /* Start of the entire buffer.  */
    unsigned int size;  /* Malloc'd size of buffer. */
    FILE *fp;           /* File, or NULL if this is an internal buffer.  */
    struct floc floc;   /* Info on the file in fp (if any).  */
  };
```

The `size` field tracks the size of the entire buffer, and `fp` is the `FILE` pointer for the input file. The `floc` structure isn't of interest for studying the routine.

The function returns the number of lines in the buffer. (The line numbers here are relative to the start of the function, not the source file.)

*static long readline(struct ebuffer *ebuf)*

```
1  static long
2  readline (ebuf)
3       struct ebuffer *ebuf;
4  {
5    char *p;
6    char *end;
7    char *start;
8    long nlines = 0;
9
10   /* The behaviors between string and stream buffers are different enough to
11      warrant different functions.  Do the Right Thing.  */
12
13   if (!ebuf->fp)
14     return readstring (ebuf);
15
16   /* When reading from a file, we always start over at the beginning of the
17      buffer for each new line.  */
18
19   p = start = ebuf->bufstart;
20   end = p + ebuf->size;
21   *p = '\0';
```

We start by noticing that GNU Make is written in K&R C for maximal portability. The initial part declares variables, and if the input is coming from a string (such as from the expansion of a macro), the code hands things off to a different function, `readstring()` (lines 13 and 14). The test '`!ebuf->fp`' (line 13) is a shorter (and less clear, in our opinion) test for a null pointer; it's the same as '`ebuf->fp == NULL`'.

Lines 19–21 initialize the pointers, and insert a NUL byte, which is the C string terminator character, at the end of the buffer. The function then starts a loop (lines 23–95), which runs as long as there is more input.

```
23    while (fgets (p, end - p, ebuf->fp) != 0)
24      {
25        char *p2;
26        unsigned long len;
27        int backslash;
28
29        len = strlen (p);
30        if (len == 0)
31          {
32            /* This only happens when the first thing on the line is a '\0'.
33               It is a pretty hopeless case, but (wonder of wonders) Athena
34               lossage strikes again!  (xmkmf puts NULs in its makefiles.)
35               There is nothing really to be done; we synthesize a newline so
36               the following line doesn't appear to be part of this line.  */
37            error (&ebuf->floc,
38                   _("warning: NUL character seen; rest of line ignored"));
39            p[0] = '\n';
40            len = 1;
41          }
```

The `fgets()` function (line 23) takes a pointer to a buffer, a count of bytes to read, and a `FILE *` variable for the file to read from. It reads one less than the count so that it can terminate the buffer with `'\0'`. This function is good since it allows you to avoid buffer overflows. It stops upon encountering a newline or end-of-file, and if the newline is there, it's placed in the buffer. It returns `NULL` on failure or the (pointer) value of the first argument on success.

In this case, the arguments are a pointer to the free area of the buffer, the amount of room left in the buffer, and the `FILE` pointer to read from.

The comment on lines 32–36 is self-explanatory; if a zero byte is encountered, the program prints an error message and pretends it was an empty line. After compensating for the NUL byte (lines 30–41), the code continues.

```
43        /* Jump past the text we just read.  */
44        p += len;
45
46        /* If the last char isn't a newline, the whole line didn't fit into the
47           buffer.  Get some more buffer and try again.  */
48        if (p[-1] != '\n')
49          goto more_buffer;
50
51        /* We got a newline, so add one to the count of lines.  */
52        ++nlines;
```

Lines 43–52 increment the pointer into the buffer past the data just read. The code then checks whether the last character read was a newline. The construct `p[-1]` (line 48) looks at the character *in front of* `p`, just as `p[0]` is the current character and `p[1]` is the next. This looks strange at first, but if you translate it into terms of pointer math, `*(p-1)`, it makes more sense, and the indexing form is possibly easier to read.

If the last character was not a newline, this means that we've run out of space, and the code goes off (with `goto`) to get more (line 49). Otherwise, the line count is incremented.

```
54  #if !defined(WINDOWS32) && !defined(__MSDOS__)
55        /* Check to see if the line was really ended with CRLF; if so ignore
56           the CR.  */
57        if ((p - start) > 1 && p[-2] == '\r')
58          {
59            --p;
60            p[-1] = '\n';
61          }
62  #endif
```

Lines 54–62 deal with input lines that follow the Microsoft convention of ending with a Carriage Return-Line Feed (CR-LF) combination, and not just a Line Feed (or newline), which is the Linux/Unix convention. Note that the `#ifdef` *excludes* the code on Microsoft systems; apparently the `<stdio.h>` library on those systems handles this conversion automatically. This is also true of other non-Unix systems that support Standard C.

```
64        backslash = 0;
65        for (p2 = p - 2; p2 >= start; --p2)
66          {
67            if (*p2 != '\\')
68              break;
69            backslash = !backslash;
70          }
71
```

```
72          if (!backslash)
73            {
74              p[-1] = '\0';
75              break;
76            }
77
78          /* It was a backslash/newline combo.  If we have more space, read
79             another line.  */
80          if (end - p >= 80)
81            continue;
82
83          /* We need more space at the end of our buffer, so realloc it.
84             Make sure to preserve the current offset of p.  */
85        more_buffer:
86          {
87            unsigned long off = p - start;
88            ebuf->size *= 2;
89            start = ebuf->buffer = ebuf->bufstart = (char *) xrealloc (start,
90                                                                       ebuf->size);
91            p = start + off;
92            end = start + ebuf->size;
93            *p = '\0';
94          }
95        }
```

So far we've dealt with the mechanics of getting at least one complete line into the buffer. The next chunk handles the case of a continuation line. It has to make sure, though, that the final backslash isn't part of multiple backslashes at the end of the line. It tracks whether the total number of such backslashes is odd or even by toggling the `backslash` variable from `0` to `1` and back. (Lines 64–70.)

If the number is even, the test '`! backslash`' (line 72) will be true. In this case, the final newline is replaced with a NUL byte, and the code leaves the loop.

On the other hand, if the number is odd, then the line contained an even number of backslash pairs (representing escaped backslashes, `\\` as in C), and a final backslash-newline combination.[5] In this case, if at least 80 free bytes are left in the buffer, the program `continue`s around the loop to read another line (lines 78–81). (The use of the magic number `80` isn't great; it would have been better to define and use a symbolic constant.)

---

[5] This code has the scent of practical experience about it: It wouldn't be surprising to learn that earlier versions simply checked for a final backslash before the newline, until someone complained that it didn't work when there were multiple backslashes at the end of the line.

Upon reaching line 83, the program needs more space in the buffer. Here's where the dynamic memory management comes into play. Note the comment about preserving `p` (lines 83–84); we discussed this earlier in terms of reinitializing pointers into dynamic memory. `end` is also reset. Line 89 resizes the memory.

Note that here the function being called is `xrealloc()`. Many GNU programs use "wrapper" functions around `malloc()` and `realloc()` that automatically print an error message and exit if the standard routines return `NULL`. Such a wrapper might look like this:

```
extern const char *myname;    /* set in main() */

void *xrealloc(void *ptr, size_t amount)
{
    void *p = realloc(ptr, amount);

    if (p == NULL) {
        fprintf(stderr, "%s: out of memory!\n", myname);
        exit(1);
    }
}
```

Thus, if `xrealloc()` returns, it's guaranteed to return a valid pointer. (This strategy complies with the "check every call for errors" principle while avoiding the code clutter that comes with doing so using the standard routines directly.) In addition, this allows valid use of the construct '`ptr = xrealloc(ptr, new_size)`', which we otherwise warned against earlier.

Note that it is not always appropriate to use such a wrapper. If you wish to handle errors yourself, you shouldn't use it. On the other hand, if running out of memory is always a fatal error, then such a wrapper is quite handy.

```
97    if (ferror (ebuf->fp))
98      pfatal_with_name (ebuf->floc.filenm);
99
100   /* If we found some lines, return how many.
101      If we didn't, but we did find _something_, that indicates we read the last
102      line of a file with no final newline; return 1.
103      If we read nothing, we're at EOF; return -1.  */
104
105   return nlines ? nlines : p == ebuf->bufstart ? -1 : 1;
106 }
```

Finally, the `readline()` routine checks for I/O errors, and then returns a descriptive return value. The function `pfatal_with_name()` (line 98) doesn't return.

3.2.1.9 GLIBC Only: Reading Entire Lines: `getline()` and `getdelim()`

Now that you've seen how to read an arbitrary-length line, you can breathe a sigh of relief that you don't have to write such a function for yourself. GLIBC provides two functions to do this for you:

GLIBC

```
#define _GNU_SOURCE 1
#include <stdio.h>
#include <sys/types.h>          /* for ssize_t */

ssize_t getline(char **lineptr, size_t *n, FILE *stream);
ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);
```

Defining the constant `_GNU_SOURCE` brings in the declaration of the `getline()` and `getdelim()` functions. Otherwise, they're implicitly declared as returning `int`. `<sys/types.h>` is needed so you can declare a variable of type `ssize_t` to hold the return value. (An `ssize_t` is a "signed `size_t`." It's meant for the same use as a `size_t`, but for places where you need to be able to hold negative values as well.)

Both functions manage dynamic storage for you, ensuring that the buffer containing an input line is always big enough to hold the input line. They differ in that `getline()` reads until a newline character, and `getdelim()` uses a user-provided delimiter character. The common arguments are as follows:

`char **lineptr`
: A pointer to a `char *` pointer to hold the address of a dynamically allocated buffer. It should be initialized to `NULL` if you want `getline()` to do all the work. Otherwise, it should point to storage previously obtained from `malloc()`.

`size_t *n`
: An indication of the size of the buffer. If you allocated your own buffer, `*n` should contain the buffer's size. Both functions update `*n` to the new buffer size if they change it.

`FILE *stream`
: The location from which to get input characters.

The functions return `-1` upon end-of-file or error. The strings hold the terminating newline or delimiter (if there was one), as well as a terminating zero byte. Using `getline()` is easy, as shown in `ch03-getline.c`:

```
/* ch03-getline.c --- demonstrate getline(). */

#define _GNU_SOURCE 1
#include <stdio.h>
#include <sys/types.h>

/* main --- read a line and echo it back out until EOF. */

int main(void)
{
    char *line = NULL;
    size_t size = 0;
    ssize_t ret;

    while ((ret = getline(& line, & size, stdin)) != -1)
        printf("(%lu) %s", size, line);

    return 0;
}
```

Here it is in action, showing the size of the buffer. The third input and output lines are purposely long, to force `getline()` to grow the buffer; thus, they wrap around:

```
$ ch03-getline                                        Run the program
this is a line
(120) this is a line
And another line.
(120) And another line.
A lllllllllllllllloooooooooooooooooooooooooooooooooonnnnnnnnnnnnnnnnnnnggggg
ggggggggg   lllliiiiiiiiiiiiiiiiiinnnnnnnnnnnnnnnnnnnneeeeeeeeee
(240) A lllllllllllllllloooooooooooooooooooooooooooooooonnnnnnnnnnnnnnnng
nnnggggggggggg   lllliiiiiiiiiiiiiiiiinnnnnnnnnnnnnnnnnnneeeeeeeeee
```

3.2.2 String Copying: `strdup()`

One extremely common operation is to allocate storage for a copy of a string. It's so common that many programs provide a simple function for it instead of using inline code, and often that function is named `strdup()`:

```
#include <string.h>

/* strdup --- malloc() storage for a copy of string and copy it */

char *strdup(const char *str)
{
    size_t len;
    char *copy;

    len = strlen(str) + 1;   /* include room for terminating '\0' */
    copy = malloc(len);

    if (copy != NULL)
        strcpy(copy, str);

    return copy;             /* returns NULL if error */
}
```

With the 2001 POSIX standard, programmers the world over can breathe a little easier: This function is now part of POSIX as an XSI extension:

```
#include <string.h>                                              XSI

char *strdup(const char *str);          Duplicate str
```

The return value is `NULL` if there was an error or a pointer to dynamically allocated storage holding a copy of `str`. The returned value should be freed with `free()` when it's no longer needed.

3.2.3 System Calls: `brk()` and `sbrk()`

The four routines we've covered (`malloc()`, `calloc()`, `realloc()`, and `free()`) are the standard, portable functions to use for dynamic memory management.

On Unix systems, the standard functions are implemented on top of two additional, very primitive routines, which directly change the size of a process's address space. We present them here to help you understand how GNU/Linux and Unix work ("under the hood" again); it is highly unlikely that you will ever need to use these functions in a regular program. They are declared as follows:

```
#include <unistd.h>                                              Common
#include <malloc.h>      /* Necessary for GLIBC 2 systems */

int brk(void *end_data_segment);
void *sbrk(ptrdiff_t increment);
```

The `brk()` system call actually changes the process's address space. The address is a pointer representing the end of the data segment (really the heap area, as shown earlier in Figure 3.1). Its argument is an absolute logical address representing the new end of the address space. It returns `0` on success or `-1` on failure.

The `sbrk()` function is easier to use; its argument is the increment in bytes by which to change the address space. By calling it with an increment of `0`, you can determine where the address space currently ends. Thus, to increase your address space by 32 bytes, use code like this:

```
char *p = (char *) sbrk(0);     /* get current end of address space */
if (brk(p + 32) < 0) {
    /* handle error */
}
/* else, change worked */
```

Practically speaking, you would not use `brk()` directly. Instead, you would use `sbrk()` exclusively to grow (or even shrink) the address space. (We show how to do this shortly, in Section 3.2.5, "Address Space Examination," page 78.)

Even more practically, you should *never* use these routines. A program using them can't then use `malloc()` also, and this is a big problem, since many parts of the standard library rely on being able to use `malloc()`. Using `brk()` or `sbrk()` is thus likely to lead to hard-to-find program crashes.

But it's worth knowing about the low-level mechanics, and indeed, the `malloc()` suite of routines is implemented with `sbrk()` and `brk()`.

3.2.4 Lazy Programmer Calls: `alloca()`

"Danger, Will Robinson! Danger!"
—**The Robot**—

There is one additional memory allocation function that you should know about. We discuss it *only* so that you'll understand it when you see it, but you should *not* use it in new programs! This function is named `alloca()`; it's declared as follows:

```
/* Header on GNU/Linux, possibly not all Unix systems */          Common
#include <alloca.h>

void *alloca(size_t size);
```

The `alloca()` function allocates `size` bytes from the *stack*. What's nice about this is that the allocated storage disappears when the function returns. There's no need to explicitly free it because it goes away automatically, just as local variables do.

At first glance, `alloca()` seems like a programming panacea; memory can be allocated that doesn't have to be managed at all. Like the Dark Side of the Force, this is indeed seductive. And it is similarly to be avoided, for the following reasons:

- The function is nonstandard; it is not included in any formal standard, either ISO C or POSIX.
- The function is not portable. Although it exists on many Unix systems and GNU/Linux, it doesn't exist on non-Unix systems. This is a problem, since it's often important for code to be multiplatform, above and beyond just Linux and Unix.
- On some systems, `alloca()` can't even be implemented. All the world is not an Intel x86 processor, nor is all the world GCC.
- Quoting the manpage (emphasis added): "The `alloca` function is machine and compiler dependent. *On many systems its implementation is buggy.* Its use is discouraged."
- Quoting the manpage again: "On many systems `alloca` cannot be used inside the list of arguments of a function call, because the stack space reserved by `alloca` would appear on the stack in the middle of the space for the function arguments."
- It encourages sloppy coding. Careful and correct memory management isn't hard; you just to have to think about what you're doing and plan ahead.

GCC generally uses a built-in version of the function that operates by using inline code. As a result, there are other consequences of `alloca()`. Quoting again from the manpage:

> The fact that the code is inlined means that it is impossible to take the address of this function, or to change its behavior by linking with a different library.
>
> The inlined code often consists of a single instruction adjusting the stack pointer, and does not check for stack overflow. Thus, there is no `NULL` error return.

The manual page doesn't go quite far enough in describing the problem with GCC's built-in `alloca()`. If there's a stack overflow, the return value is *garbage*. And you have no way to tell! This flaw makes GCC's `alloca()` impossible to use in robust code.

All of this should convince you to stay away from `alloca()` for any new code that you may write. If you're going to have to write portable code using `malloc()` and `free()` anyway, there's no reason to also write code using `alloca()`.

3.2.5 Address Space Examination

The following program, `ch03-memaddr.c`, summarizes everything we've seen about the address space. It does many things that you should not do in practice, such as call `alloca()` or use `brk()` and `sbrk()` directly:

```
1  /*
2   * ch03-memaddr.c --- Show address of code, data and stack sections,
3   *                    as well as BSS and dynamic memory.
4   */
5
6  #include <stdio.h>
7  #include <malloc.h>     /* for definition of ptrdiff_t on GLIBC */
8  #include <unistd.h>
9  #include <alloca.h>     /* for demonstration only */
10
11 extern void afunc(void);    /* a function for showing stack growth */
12
13 int bss_var;            /* auto init to 0, should be in BSS */
14 int data_var = 42;      /* init to nonzero, should be data */
15
16 int
17 main(int argc, char **argv) /* arguments aren't used */
18 {
19     char *p, *b, *nb;
20
21     printf("Text Locations:\n");
22     printf("\tAddress of main: %p\n", main);
23     printf("\tAddress of afunc: %p\n", afunc);
24
25     printf("Stack Locations:\n");
26     afunc();
27
28     p = (char *) alloca(32);
29     if (p != NULL) {
30         printf("\tStart of alloca()'ed array: %p\n", p);
31         printf("\tEnd of alloca()'ed array: %p\n", p + 31);
32     }
33
```

```
34      printf("Data Locations:\n");
35      printf("\tAddress of data_var: %p\n", & data_var);
36
37      printf("BSS Locations:\n");
38      printf("\tAddress of bss_var: %p\n", & bss_var);
39
40      b = sbrk((ptrdiff_t) 32);   /* grow address space */
41      nb = sbrk((ptrdiff_t) 0);
42      printf("Heap Locations:\n");
43      printf("\tInitial end of heap: %p\n", b);
44      printf("\tNew end of heap: %p\n", nb);
45
46      b = sbrk((ptrdiff_t) -16);  /* shrink it */
47      nb = sbrk((ptrdiff_t) 0);
48      printf("\tFinal end of heap: %p\n", nb);
49  }
50
51  void
52  afunc(void)
53  {
54      static int level = 0;       /* recursion level */
55      auto int stack_var;         /* automatic variable, on stack */
56
57      if (++level == 3)           /* avoid infinite recursion */
58          return;
59
60      printf("\tStack level %d: address of stack_var: %p\n",
61              level, & stack_var);
62      afunc();                    /* recursive call */
63  }
```

This program prints the locations of the two functions `main()` and `afunc()` (lines 22–23). It then shows how the stack grows downward, letting `afunc()` (lines 51–63) print the address of successive instantiations of its local variable `stack_var`. (`stack_var` is purposely declared `auto`, to emphasize that it's on the stack.) It then shows the location of memory allocated by `alloca()` (lines 28–32). Finally it prints the locations of data and BSS variables (lines 34–38), and then of memory allocated directly through `sbrk()` (lines 40–48). Here are the results when the program is run on an Intel GNU/Linux system:

```
$ ch03-memaddr
Text Locations:
    Address of main: 0x804838c
    Address of afunc: 0x80484a8
Stack Locations:
    Stack level 1: address of stack_var: 0xbffff864
    Stack level 2: address of stack_var: 0xbffff844     Stack grows downward
    Start of alloca()'ed array: 0xbffff860
    End of alloca()'ed array: 0xbffff87f                Addresses are on the stack
```

```
Data Locations:
    Address of data_var: 0x80496b8
BSS Locations:
    Address of bss_var: 0x80497c4          BSS is above data variables
Heap Locations:
    Initial end of heap: 0x80497c8         Heap is immediately above BSS
    New end of heap: 0x80497e8             And grows upward
    Final end of heap: 0x80497d8           Address spaces can shrink
```

3.3 Summary

- Every Linux (and Unix) program has different memory areas. They are stored in separate parts of the executable program's disk file. Some of the sections are loaded into the same part of memory when the program is run. All running copies of the same program share the executable code (the text segment). The `size` program shows the sizes of the different areas for relocatable object files and fully linked executable files.
- The address space of a running program may have holes in it, and the size of the address space can change as memory is allocated and released. On modern systems, address `0` is not part of the address space, so don't attempt to dereference `NULL` pointers.
- At the C level, memory is allocated or reallocated with one of `malloc()`, `calloc()`, or `realloc()`. Memory is freed with `free()`. (Although `realloc()` can do everything, using it that way isn't recommended). It is unusual for freed memory to be removed from the address space; instead, it is reused for later allocations.
- Extreme care must be taken to
  - Free only memory received from the allocation routines,
  - Free such memory once and only once,
  - Free unused memory, and
  - Not "leak" any dynamically allocated memory.
- POSIX provides the `strdup()` function as a convenience, and GLIBC provides `getline()` and `getdelim()` for reading arbitrary-length lines.

- The low-level system call interface functions, `brk()` and `sbrk()`, provide direct but primitive access to memory allocation and deallocation. Unless you are writing your own storage allocator, you should not use them.
- The `alloca()` function for allocating memory on the stack exists, but is not recommended. Like being able to recognize poison ivy, you should know it only so that you'll know to avoid it.

Exercises

1. Starting with the structure—

   ```
   struct line {
       size_t buflen;
       char *buf;
       FILE *fp;
   };
   ```

   —write your own `readline()` function that will read an any-length line. Don't worry about backslash continuation lines. Instead of using `fgets()` to read lines, use `getc()` to read characters one at a time.
2. Does your function preserve the terminating newline? Explain why or why not.
3. How does your function handle lines that end in CR-LF?
4. How do you initialize the structure? With a separate routine? With a documented requirement for specific values in the structure?
5. How do you indicate end-of-file? How do you indicate that an I/O error has occurred? For errors, should your function print an error message? Explain why or why not.
6. Write a program that uses your function to test it, and another program to generate input data to the first program. Test your function.
7. Rewrite your function to use `fgets()` and test it. Is the new code more complex or less complex? How does its performance compare to the `getc()` version?
8. Study the V7 *end*(3) manpage (`/usr/man/man3/end.3` in the V7 distribution). Does it shed any light on how '`sbrk(0)`' might work?
9. Enhance `ch03-memaddr.c` to print out the location of the arguments and the environment. In which part of the address space do they reside?