CSC 352, Fall 2015
Assignment 11
Due: Wednesday, November 25 at 23:59:59

A little bit of **make**

make is a program that's used to build executables and more based on specifications in a "makefile". We'll be talking about make soon, and I decided to have the a11 tester use make to build executables. In order for things to work you need to make a symlink in your assignment 11 directory to a11/Makefile, like this:

```
ln -s a11/Makefile .
```

With that symlink in place you should be able to build an executable for lam (problem 1) by simply typing "make lam", like this:

```
% make lam
gcc -Werror -Wall -g -std=c1x -I/cs/www/classes/cs352/fall15/h -o
lam lam.c
```

make uses a combination of built-in rules and rules specified in the makefile to decide what needs to be done. For example, if after doing "make lam" I do it again immediately, I see this:

```
% make lam
make: `lam' is up to date.
```

That's because that make knows that lam "depends on" lam.c, and if lam is newer than lam.c, there shouldn't be any reason to rebuild it.

Using make should largely free you from needing to do compilation by hand with gcc.

My make skills are pretty rusty and a11/Makefile is correspondingly simplistic, but I'll be brushing up on make.

File I/O and operations on files

One of the assignment's four programs, lam, opens files and reads from them. A second program, mar, performs a number of file-related operations: opening and creating files, reading and writing files, and getting information like file type and size. Following the section in the slides on structures, we'll have a section on file handling where we talk about all these things.

We'll also see how to use getline(3) to read lines of any length, but you'll see it's not as convenient as Java's BufferedReader.readLine().

Those file-handling slides will be distributed on Friday, November 13, but we might not get through them all until Wednesday, November 16. In the meantime, you can look ahead in the slides and/or get started with mcycle.c and alloc.c.

No `valgrind`-clean requirement

I think this assignment is hard enough without the additional requirement of `valgrind` finding no errors, so the tester doesn't use `valgrind`, but I recommend that you use `valgrind` as part of your development process.

The Usual Stuff

All the usual stuff on assignments applies to this one, too: make an `a`*`N`* symlink, use our `gcc` alias, use the Tester, the Tester runs `gcc` first, assume `malloc` never return 0, etc. Refer to previous write-ups if you need a refresher.

No restrictions, but...

There are no restrictions on your solutions for this assignment; you're free to use any elements of the C language and library routines that you wish. However, the assignment is written with the intention of being solvable using only what you've seen on the slides, the previous assignments, and additional routines mentioned in this write-up, like `strtok` for `mcycle.c`. If you find yourself going beyond that set of things, you're probably overlooking a simpler solution, and/or making a problem harder than intended. And, you might not be getting practice with the things I'd like you to be getting practice with.

Probably more tests coming

A set of tests for this assignment is in place but I'd like to improve it. I'm giving myself through 3:00pm on Wednesday, November 18 to do so.

Problem 1. (15 points) `lam.c`

For this problem you are to write a C program named `lam` that is a simplified version of the once-common `lam` standard utility.

"`lam`" stands for "laminate". It displays the contents of two or more files side by side, separating lines with a separator that's specified as `lam`'s first argument. Here are two files to work with:

```
% cat a11/nums
one
two
three
four
five
% cat a11/lam.1
10
20
30
40
%
```

Let's laminate them, with three dots as a separator:

```
% lam ... a11/nums a11/lam.1
one...10
two...20
three...30
four...40
%
```

A slightly difficult part of the problem is this: `lam` stops whenever one of the files runs out. As we can see above, `lam` stopped after four lines because `a11/lam.1` had only four lines. (The shortest file line-wise "rules".)

Let's combine `lam` with a bash facility called *process substitution* to number lines of `cal` output:

```
% lam ". " <(seq 100) <(cal)
1.    November 2015
2. Su Mo Tu We Th Fr Sa
3.  1  2  3  4  5  6  7
4.  8  9 10 11 12 13 14
5. 15 16 17 18 19 20 21
6. 22 23 24 25 26 27 28
7. 29 30
8.
```

The construction `<(`*`command-line`*`)` is process substitution: it says to run the specified `command-line` and make it's output as file, which is what `lam` wants for arguments. Try putting an "`echo`" in front of "`lam`" to see a little more of what bash is doing.

In the above case, `seq 100` would produce 100 lines but because `cal` produced only eight lines, `lam` produces only eight lines of output.

Any number of files can be specified on the command line. Here's an example with four files:

```
% lam - a11/nums a11/delivs <(seq 10) a11/lam.1
one-lam.c-1-10
two-mcycle.c-2-20
three-mar.c-3-30
four-alloc.c-4-40
```

Above the write-up says that `lam` displays "two or more files side by side" but in fact `lam` is happy with one file, too, although never using the separator.

```
% lam - a11/nums
one
two
three
four
five
```

`lam` can handle files with any number of lines of any length. `lam` does no error handling whatsoever.

Problem 2. (20 points) `mcycle.c`

For this problem you are to write a very simple macro processor. `mcycle` reads lines from standard input and writes lines to standard output. Here's a sample input file:

```
% cat a11/mcycle.1
color=red,green,blue
what=pencil,crayon
   The <color> <what> made a <color> mark that the <color>
<what> erased.  Then the <what> became a <what>.
```

The input has two sections. The first is a series of "variables", consisting of a name, an equals sign, and one or more values, separated by commas. The second is a text section, a series of lines into which the values of the variables are to be cyclically substituted. Text enclosed in angle brackets is assumed to be a variable name. Here is the result of running `mcycle`:

```
% mcycle < a11/mcycle.1
   The red pencil made a green mark that the blue
crayon erased.  Then the pencil became a crayon.
```

Each value of a variable is substituted in turn, possibly cycling through the values of a variable many times.

The transition between the variable section and the text section is indicated by the absence of an equals sign on a line.

If the substitution for a variable is the pound sign then it indicates that a usage count, starting at 1, is to be inserted instead:

```
% cat a11/mcycle.3
N=#
what=a,b
<what><N><what><N><what><N><what><N><what><N>
% mcycle < a11/mcycle.3
a1b2a3b4a5
```

If a # appears it will be the only character following the equal sign. No whitespace will appear on a variable specification line.

You may assume there will be no more than 100 variables and that a variable will have no more than 50 values to cycle through, but you may not assume any maximum length for values. The text section may contain any number of lines, of any length. (Use `getline(3)` to read lines.)

You may assume that text enclosed in angle brackets is always a variable name. A variable name is never split across lines. Variable references are never nested (e.g. `<x<y>>`).

Implementation Notes

One way to approach this problem is to use an array of structures:

```
#define MAX_VARS 100
#define MAX_VALS 50
struct variable {
    char *name;
    char *reps[MAX_VALS];
    int count;
    };

struct variable variables[MAX_VARS];
```

Two string-handling routines used in my solution are `strchr` and `strtok`.

Problem 3. (35 points) `mar.c`

In the UNIX world, files that are actually collections of files are commonly called "archives". If you do `man -k archive`, you'll see dozens of programs that work with "archives". We used `jar`, the Java archive tool, to look at some Java archive files, which are collections of `.class` files and more. The `aN/turnin` scripts use `tar` to create a time-stamped "tar file" that's then submitted using `turnin(1)`. Zip files are archives.

In this problem you are to write a simple file archiving utility. It has so few capabilities that it is called `mar`, for micro-archiver.

`mar` has three modes of operation: (1) create an archive, (2) show the table of contents for an archive, and (3) extract files from an archive. Here are the general forms of invocation for the three modes:

```
mar c ARCHIVE FILE1 ... FILEN
mar t ARCHIVE
mar x ARCHIVE [FILE1 ... FILEN]
```

Here are three files to work with:

```
% ls -l empty nums lets
-rw-rw-r-- 1 whm whm  0 Nov 12 04:20 empty
-rw-rw-r-- 1 whm whm  6 Nov 12 04:20 lets
-rw-rw-r-- 1 whm whm 24 Nov 12 04:20 nums
% cat lets
a
b
c
% cat nums
one
two
three
four
five
```

Let's make an archive named `a.mar` containing those three files:

```
% mar c a.mar lets nums empty
Added lets
Added nums
Added empty
```

Let's see what `a.mar` contains:

```
% cat a.mar
#-h- 6 lets
a
b
c
#-h- 24 nums
one
two
three
four
five
#-h- 0 empty
% ls -l a.mar
-rw-rw-r-- 1 whm whm 68 Nov 12 04:32 a.mar
```

Each file in a `mar` archive is preceded by a header line that consists of "`#-h- `" followed by one blank, the file size in bytes, a second blank, the file name, and a newline. (`mar` cannot handle file names that contain a newline.)

Let's see the table of contents for `a.mar`:

```
% mar t a.mar
lets (6 bytes)
nums (24 bytes)
empty (0 bytes)
```

Because `mar` goes from file to file based only on length, rather than simply looking for lines that start with "`#-h-`", `mar` archives can contain arbitrary files, including other `mar` archives. Example:

```
% mar c b.mar lets a.mar nums
Added lets
Added a.mar
Added nums
% ls -l b.mar
-rw-rw-r-- 1 whm whm 137 Nov 12 04:52 b.mar
% mar t b.mar
lets (6 bytes)
a.mar (68 bytes)
nums (24 bytes)
```

Files to archive can be specified with an arbitrary path:

```
% mar c x.mar /etc/passwd a11/../a10/picklines a11/delivs
Added /etc/passwd
Added a11/../a10/picklines
Added a11/delivs
```

```
% mar t x.mar
/etc/passwd (228847 bytes)
a11/../a10/picklines (11208 bytes)
a11/delivs (46 bytes)
```

Note: if you're working through these examples you might see a different size for `/etc/passwd`—it changes over time, of course.

`mar` can only archive regular files. If it encounters anything else among its arguments, it skips it, noting that with a message on `stderr`. In the following invocation I try to archive my `a11` symlink and a directory named `backup`, along with all my C files:

```
% mar c c.mar *.c a11 backup
Added alloc.c
Added lam.c
Added mar.c
Added mcycle.c
a11: skipped
backup: skipped
```

Note that `mar` knows nothing about that `*.c` wildcard—that's expanded by bash.

`mar` stores path information but `mar` does not support creation of trees—files are always extracted into the current directory. Let's go into a new directory and try an extraction:

```
% mkdir work
% cd work
% mar t ../x.mar
/etc/passwd (228847 bytes)
a11/../a10/picklines (11208 bytes)
a11/delivs (46 bytes)
% mar x ../x.mar
Extracted passwd
Extracted picklines
Extracted delivs
% ls -l
total 2
-rw-rw-r-- 1 whm whm     46 Nov 12 05:05 delivs
-rw-rw-r-- 1 whm whm 228847 Nov 12 05:05 passwd
-rw-rw-r-- 1 whm whm  11208 Nov 12 05:05 picklines
```

If a file being extracted already exists, `mar` simply overwrites it without warning.

Recall the general form of `mar x` shown above:

```
mar x ARCHIVE [FILE1 ... FILEN]
```

By default, `mar x` extracts all files, but the archive name can be followed by the names of one or more files to extract. Still in the directory `work`, let's extract only `nums` and `lets` from `a.mar`:

```
% mar x ../a.mar nums lets dates
```

```
Extracted lets
Extracted nums
```

If a file is named that's not in the archive, such as `dates` in the above example, it is silently ignored.

Error handling

Invalid invocations produce a usage message. A sampling of errors follows. The tester will exercise all errors you are expected to handle.

Mode is not `c`, `t`, or `x`:

```
% mar a ../c.mar
Usage: mar [ctx] FILE [FILES...]
```

Non-existent archive:

```
% mar x /c.mar
/c.mar: No such file or directory
```

With a non-existent file note that the file is skipped, but execution is not terminated:

```
% mar c x.mar nums /etc/psword lets
Added nums
/etc/psword: No such file or directory
Added lets
```

Problem 4. (40 points) alloc.c

This problem builds on the memory allocator developed in assignment 10. Three new features are to be supported: multiple memory pools, block overrun/underrun checking, and "scribbling".

One of the limitations of the assignment 10 allocator is that there is a single pool that holds a fixed number of fixed-size blocks. In this allocator there may be many pools of memory. One of the new calls is `add_pool`:

```
void add_pool(int nblocks, int block_size);
```

This requests that a pool containing `nblocks` of `block_size` bytes be created. A user might create several pools:

```
add_pool(1000, 32); // 1000 blocks of 32 bytes
add_pool(100, 1000);     // 100 blocks of 1000 bytes
add_pool(500, 256); // 500 blocks of 256 bytes
```

The function `show_pool(char *label)` has been replaced with `show_pools(char *label)`—note the plural. Here's what `show_pools("The pools:")` produces after the above `add_pool` calls.

```
The pools:
```

```
---
Pool 1: 1000 blocks of 32 bytes
Total: 0 allocated blocks, 0 allocated bytes
---
Pool 2: 500 blocks of 256 bytes
Total: 0 allocated blocks, 0 allocated bytes
---
Pool 3: 100 blocks of 1000 bytes
Total: 0 allocated blocks, 0 allocated bytes
---
Total for all pools: 0 allocated blocks, 0 allocated bytes
```

Note that the pools are shown in order by block size, smallest first.

The function `alloc_block` has the same interface as before but it returns a block from the pool with the smallest block size that can accommodate the request.

Here's example of a sequence of allocations: (`a11/alloc1.c`)

```
int main()
{
    add_pool(3, 32);    // Note: 3 blocks of 32 bytes
    add_pool(100, 1000);
    add_pool(500, 256);

    char *p1 = alloc_block(1000, "A");
    p1 = alloc_block(100, "B");

    for (int i = 1; i <= 5; i++) {
        p1 = alloc_block(10, "C");
        p1 = p1; // avoid a "p1 unused" warning
        }

    show_pools("After allocations:");
}
```

Output:

```
After allocations:
---
Pool 1: 3 blocks of 32 bytes
Block 0: 10 bytes at 0x0xf91090, tag: "C"
Block 1: 10 bytes at 0x0xf910c0, tag: "C"
Block 2: 10 bytes at 0x0xf910f0, tag: "C"
Total: 3 allocated blocks, 30 allocated bytes
---
Pool 2: 500 blocks of 256 bytes
Block 0: 100 bytes at 0x0x7f439650b010, tag: "B"
Block 1: 10 bytes at 0x0x7f439650b120, tag: "C"
Block 2: 10 bytes at 0x0x7f439650b230, tag: "C"
Total: 3 allocated blocks, 120 allocated bytes
---
```

```
Pool 3: 100 blocks of 1000 bytes
Block 0: 1000 bytes at 0x0xf91640, tag: "A"
Total: 1 allocated blocks, 1000 allocated bytes
---
Total for all pools: 7 allocated blocks, 1150 allocated bytes
```

Note that two 10-byte allocations came from pool 2. That's because pool 1, with 32-byte blocks, was full.

<u>The second new feature</u> to be added in this version of the allocator is checking for block overruns and underruns using eight-byte "guard zones" before and after the memory reserved for the user. The idea is simple: `alloc_block(N, ...)`, reserves 8+N+8 bytes and returns the address of the 9th byte. The guard zones are filled with eight `'G'`s. The presence of a value other than `'G'` in any of the guard bytes indicates that there has perhaps been an overrun or underrun of the block. In any event, a non-`'G'` indicates that some line of code has stored a value in the wrong place.

<u>The third new feature</u>, "scribbling", is best described along with the guard zones. The idea of scribbling is simple: write "unexpected" values into memory whose contents are indeterminate. For example, the memory allocated by `alloc_block` is uninitialized—it can contain any value whatsoever. Certain values in that memory—such as zeros or valid memory addresses—can lead to lucky programs. Along with filling the guard zones with `'G'`s, `alloc_block` fills the user's portion of the memory with `'U'`s, for "uninitialized".

Consider this call: `p = alloc_block(3, ...)`. Here's the result:

```
GGGGGGGGUUUGGGGGGGG
        ^
        |
        p
```

We might see this with `gdb`:

```
(gdb) p p[-8]@25
$1 = "GGGGGGGGUUUGGGGGGGG\000\000\000\000\000"
```

Because the memory after the last guard byte is not assigned a value in `alloc_block`, we might see some leftover values, not just zeros:

```
(gdb) p [-8]@25
$2 = "GGGGGGGGUUUGGGGGGGGá~úq\000"
```

Next, a copy is done: `strcpy(p, "ab")`. Here's the result, <u>with an @ used to indicate a zero-valued byte</u>:

```
GGGGGGGGab@GGGGGGGG
        |
        p
```

When `free_block` is called, it writes `'F'`s (for "freed") over the user portion of the relinquished memory. After the call `free_block(p, ...)`, this is the result:

```
GGGGGGGGFFFGGGGGGGG
        |
        p
```

If the user then does `puts(p)` (a used-after-freed error!), here is the output they'd see:

```
FFFGGGGGGGG <maybe more bytes, until a zero is reached>
```

In short, `alloc_block` scribbles `'U'`s onto the user's memory and `free_block` scribbles `'F'`s.

With scribbling in hand, let's further consider guard zones. Again consider `p = alloc_block(3, ...)` and the result:

```
GGGGGGGGUUUGGGGGGGG
        |
        p
```

Let's execute some bad code:

```
strcpy(p, "abc");   // Trailing zero overruns block

int *ip = (int*)p;
ip[-1] = 0;         // Puts an int zero before the block
```

Here's the result:

```
GGGG@@@@abc@GGGGGGG // Remember: @ for zero bytes...
        |
        p
```

Here's what `gdb` might show: (remember that values past the last guard byte are undefined)

```
(gdb) p p[-8]@25
$1 = "GGGG\000\000\000\000abc\000GGGGGGG\000\000\000\000\000"
```

It would be ideal to catch overrun and underrun errors when they happen, e.g. in the `strcpy` call, but that's far beyond the scope of this simple allocator. Instead, a block or blocks are scanned for errors in three situations: when `free_block` is called, when `show_pools` is called, and when a new function, `check_blocks(char *label)`, is called. `check_blocks` scans all blocks in all pools and prints information about blocks that show evidence of corruption. Example: (`a11/alloc2.c`)

```
int main()
{
    char *p = alloc_block(3, "p");

    check_blocks("After alloc_block:");

    strcpy(p, "abc");

    check_blocks("After strcpy:");
```

```
        int *ip = (int*)p;
        ip[-1] = 0;

        check_blocks("After ip[-1] = 0:");

        show_pools("Pools:");

        puts("ready to free...");
        free_block(p, "p");
    }
```

The output follows, with blank lines inserted to separate the several segments of output. Note that several lines have wrapped around.

```
After alloc_block:

After strcpy:
Pool 1, block 0: 3 bytes at 0x0x7ffb9efde010, tag: "p" OVERRUN
BLOCK
After ip[-1] = 0:
Pool 1, block 0: 3 bytes at 0x0x7ffb9efde010, tag: "p" UNDERRUN
and OVERRUN BLOCK

Pools:
---
Pool 1: 10000 blocks of 1024 bytes
Block 0: 3 bytes at 0x0x7ffb9efde010, tag: "p" UNDERRUN and
OVERRUN BLOCK
Total: 1 allocated blocks, 3 allocated bytes
---
Total for all pools: 1 allocated blocks, 3 allocated bytes

ready to free...
free_block(0x0x7ffb9efde018, p): UNDERRUN and OVERRUN BLOCK
```

The first call, `check_blocks("After alloc_block:")` produces no output other than the label. The second call detects that there has been an overrun. By the time of the third `check_blocks` call, the block has been both overrun and underrun.

Here are specifications for the functions you are to write:

```
  void add_pool(int N, int block_size)
```

Creates a memory pool that consists of `N` blocks of length `block_size`.. The `block_size` does not include the guard zones. `N` must be greater than zero; `block_size` must be greater than zero and divisible by 8. (If not, execution is terminated with an error message; you can see it in my `add_pool` code below.) It is assumed no two pools will have the same block size.

A pool can be added at any time. **There is no limit on the number of pools that may be created.**

```
void *alloc_block(int nbytes, char *tag)
```

Allocates a block of memory capable of holding `nbytes`. The block is allocated in the pool that has the smallest block size that is capable of holding the block. Eight byte guard zones, filled with `'G'`s, are placed on each side of the `nbytes` of memory reserved for use by the caller. The `nbytes` of memory for the user (the code that calls `alloc_block`), which is between the guard zones, is filled with `'U'`s. The address returned is the address of the first `'U'`, not the first `'G'`.

`tag` is associated with the block, just like the first version of the allocator.

If no block of `nbytes` is available in any pool, `alloc_block` returns 0.

If `alloc_block` is called before there has been an `add_pool` call, `alloc_block` begins by calling `add_pool(10000, 1024)`. A way to think about it is this: if the user doesn't explicitly request a pool, they get a pool of ten thousand 1024-byte blocks. Subsequent `add_pool` calls are allowed.

```
void free_block(void *addr, char *tag)
```

The block at address `addr` is freed. Just as in the assignment 10 version, tests are performed to be sure that `addr` is an address returned by `alloc_block` and that the block is currently allocated. If so, then `free_block` checks to be sure that the guard zones are intact. If any errors are detected, a message is printed and `free_block` returns immediately. If no errors are detected then the block is freed and the caller's portion of the memory, which was filled with `'U'`s by `alloc_block`, is filled with `'F'`s.

```
void show_pools(char *label)
```

After printing `label`, `show_pools` displays information about each pool and the allocated blocks in each pool. The pools are displayed in increasing order by block size. If there is evidence of underrun and/or overrun in a block, that is included in the per-block information.

```
void check_blocks(char *label)
```

After printing `label`, `check_blocks` checks each block in each pool for evidence of underrun and/or overrun. Other than the label, which is always printed, `check_blocks` produces output only when a corrupted block is detected.

NOTE: Consult the reference versions and/or this write-up to determine the precise format of output produced by `free_block`, `check_blocks`, and `show_pool`.

Reference versions and testing

A reference version, in the form of an object file, is in `a11/alloc.o`. A currently small collection of test programs can be found in `a11`. Note that due to scribbling, some of the programs, such as `alloc2`, generate a segmentation fault at a certain point when they are working correctly.

`a11/alloc-shell` is a simple "shell" for interacting with the allocator. Just as a UNIX shell gives us a way to execute commands, `alloc-shell` gives us a way to interact directly with the allocator. It provides a collection of simple commands that map directly to the allocator functions. For example, the command "`p 10 32`" generates a call to `add_pool(10,32)`.

The source code of `a11/alloc-shell.c` is the only documentation for it but here's an example of use; user input is underlined and in bold:

```
% a11/alloc-shell

alloc> p 3 32

alloc> s
pools:
---
Pool 1: 3 blocks of 32 bytes
Total: 0 allocated blocks, 0 allocated bytes
---
Total for all pools: 0 allocated blocks, 0 allocated bytes

alloc> a 20
20 bytes at 0x155bdf8 (#0)

alloc> d 0
Dumping block at 0x155bdf8
Leading guard zone:
    0:  71  71  71  71  71  71  71  71
Block contents:
    0:  85  85  85  85  85  85  85  85  85  85
   10:  85  85  85  85  85  85  85  85  85  85
Trailing guard zone:
    0:  71  71  71  71  71  71  71  71

alloc> a 30
30 bytes at 0x155be28 (#1)

alloc> a 40
40 bytes at (nil) (#2)

alloc> f 1

alloc> d 1
Dumping block at 0x155be28
Leading guard zone:
    0:  71  71  71  71  71  71  71  71
Block contents:
    0:  70  70  70  70  70  70  70  70  70  70
   10:  70  70  70  70  70  70  70  70  70  70
   20:  70  70  70  70  70  70  70  70  70  70
Trailing guard zone:
    0:  71  71  71  71  71  71  71  71

alloc> s
```

```
pools:
---
Pool 1: 3 blocks of 32 bytes
Block 0: 20 bytes at 0x155bdf0, tag: "a cmd"
Total: 1 allocated blocks, 20 allocated bytes
---
Total for all pools: 1 allocated blocks, 20 allocated bytes
```

The file `a11/as.1` is a script for `alloc1.c`. Additional scripts may appear, and `alloc-shell` may grow more elaborate.

Implementation notes

A significant part of this problem is accommodating an arbitrary number of pools with per-pool values for block size and block counts. My solution uses a linked list of `pool_info` structures:

```
struct pool_info {
    struct pool_config config;
    int *sizes;
    char **tags;
    char *memory;
    char *memend;   // addr of first byte after pool
    struct pool_info *next;
    };
```

`malloc` is used to allocate memory for each `pool_info` structure. Additionally, memory is allocated for an array of block sizes (`pool_info.sizes`), an array of tags (`pool_info.tags`), and the memory for the pool itself (`pool_info.memory`).

Recall that blocks are allocated in the pool that has the smallest block size that is capable of holding the block. This led me to maintain the linked list of `pool_info` structures in order by block size.

`pool_info.config` is a structure that contains sizing information about the pool:

```
struct pool_config {
    int max_allocs;
    int user_block_size;
    int real_block_size;
    };
```

`max_allocs` and `user_block_size` simply hold the sizing information specified by `add_pool`. I chose to have a third member, `real_block_size`, that holds the size of the blocks with the size of the guards included. For example, given `add_pool(100, 32)`, `user_block_size` is 32 and `real_block_size` is 48 (8+32+8).

Note: In the above narrative, `pool_config` is shown after `pool_info` but `pool_config` appears first in the source code. That's required by C because `pool_info` contains a `pool_config`.

Here is the start of my `add_pool` function:

```
      void add_pool(int nblocks, int block_size)
      {
          struct pool_info *new = malloc(sizeof(struct pool_info));

          struct pool_config *config = &new->config;

          config->max_allocs = nblocks;
          config->user_block_size = block_size;
          config->real_block_size = block_size + GUARD_LENGTH * 2;

          if (nblocks <= 0 || block_size <= 0 ||
                  block_size % BLOCK_ALIGN != 0) {
              printf("invalid call: add_pool(%d, %d)\n",
                  nblocks, block_size);
              exit(1);
              }

          new->sizes = calloc(config->max_allocs, sizeof(int));
          new->tags = malloc(config->max_allocs * sizeof(char *));

          int nbytes = config->max_allocs * config->real_block_size;
          new->memory = malloc(nbytes * sizeof(char));
          new->memend = new->memory + nbytes;
          new->next = 0;
```

You are NOT required to follow the approach described above but you may find it to be a good starting point.

A pitfall in this problem is to mix up the block size requested in `add_pool`, 32 for example, and the length of the blocks held in the pool, which includes the space for the guard zones—16 more bytes than requested in `add_pool`. My solution initially used a single member, `block_size`, and adjusted for the guard zones lengths when appropriate, but I found that to be error-prone. I now use separate members in `pool_config` (`user_block_size` and `real_block_size`).

The library function `memset` is handy for the scribbling in `alloc_block` and `free_block`; `memcmp` is handy for checking the guard zones.

Problem 5. Extra Credit **`observations.txt`**

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one `"Hours:"` line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a11/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `a`*`N.YYYYMMDD.HHMMSS`*`.tz` You can run `a11/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `352f15` if you need to recover a file—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a11/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
% wc $(grep -v txt < a11/delivs)
   65   144  1266 lam.c
   95   243  2395 mcycle.c
  164   378  3897 mar.c
  263   806  7382 alloc.c
  587  1571 14940 total

% for i in $(grep -v txt a11/delivs); do echo $i: $(tr -dc \; <
$i | wc -c); done
lam.c: 31
mcycle.c: 51
mar.c: 73
alloc.c: 115
```

There are few comments in my code.

Miscellaneous

This assignment is based on the material on C slides 1-451, plus probably 2-3 more slides, yet to be written, that talk about `fseek()`.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that a student who has only taken CSC 127A and 127B but done well in them, has completed all previous assignments, knows how to use `gdb`, and has done the required reading will need 12-18 hours to complete this assignment.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems.

If you put twelve hours into this assignment and don't seem to be close to completing it, it's probably time to touch base with us. Specifically mention that you've reached twelve hours. Give us a chance to speed you up!

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)