CSC 352, Fall 2015
Assignment 10
Due: Friday, November 13 at 23:59:59

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, etc. Refer to previous write-ups if you need a refresher.

Assume `malloc` never returns 0

`malloc` returns `NULL` (`0`) if there isn't enough memory available to meet an allocation request. With modern systems that's relatively rare, but a production application should have some way to handle that possibility. However, we won't worry about it on this assignment. Simply assume that `malloc` never returns 0.

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 slides 1-397, the previous assignments, and additional routines mentioned in this write-up, like `sscanf` for `picklines.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.

Maybe more tests coming

A set of tests for this assignment but I'd like to improve it. I'm giving myself through 3pm on Monday, November 16 to do so.

Problem 1. (12 points) `warmup.c`

The purpose of this problem is to get you warmed-up by implementing several very simple functions that do memory allocation.

No full examples of operation of the functions are included in this write-up, but you can examine `a10/warmup1.c`, and its output, to clarify how things work. Use `gcc warmup.c a10/warmup1.c` for experimentation.

The grading set for this problem will be only `a10/warmup1.c`. It won't change after 3:00pm on Friday, November 6, so there's no need to worry about cases not exercised in `a10/warmup1.c`.

Below are the functions you are to implement. Prototypes for all are in `a10/warmup.h`.

`char *dup(char *s)`
returns a pointer to an allocated string that is a copy of the string `s`. There's a library function named `strdup` that does the same thing but, needless to say, you may not use `strdup` in your `dup` solution.

`int *trio(int value)`
returns a pointer to an allocated array of three `int`s that hold `value-1`, `value`, and `value+1`, respectively.

`char *char_range(char from, char to)`
returns a pointer to an allocated string that consists of the characters from `from` through `to`, inclusive. If `from` is greater than `to`, the string is empty.

`int *reverse(int *first, int *last)`
returns a pointer to an allocated array of `int`s that contains the `int`s between `first` and `last`, inclusive, in reversed order. Assume `first` <= `last`.

`int *merge(int *a, int *b, int nvals)`
merges the `int` sequences starting at `a` and `b`, both assumed to have `nvals` values, into a single sequence in an allocated array `result`, simply alternating between values in `a` and `b`. The address of the allocated array is returned. Assume `nvals >= 0`.

Example: If `nvals` is 4, `a` is `{1,2,3,4}` and `b` is `{10,20,30,40}`, the allocated array would contain `{1,10,2,20,3,30,4,40}`.

`void split(int *vals, int nvals, int **result)`
is the counterpart of merge, writing `nvals` values from `vals` alternately into two allocated arrays of `int`s that are both `nvals/2` elements in length.

`result` is the address of an uninitialized array of two `int` pointers. `split` stores the addresses of the two allocated blocks in the two elements of `result`.

Example: If `nvals` is 6 and `vals` is `{1,2,3,4,5,6}`, then the first allocated block would hold `{1,3,5}`, and the second allocated block would hold `{2,4,6}`. Assume `nvals >= 0` and is an even number.

Problem 2. (2 points) `getnth.c`

In this problem you are to implement a revised version of `getnth` from assignment 8. Here is the new prototype:

```
char *getnth(char *s, int N)
```

Instead writing the result into a buffer supplied by the user, this version returns a pointer to a string in allocated memory

If `N` is negative or too large, `getnth` returns 0. `getnth` assumes that the string `s` is well-formed

Problem 3. (5 points) `split.c`

Write a C function `char **split(char *s)` that breaks a `getnth`-style string into its components. `split` returns a pointer to an allocated array of pointers to allocated strings. The last element in the array of pointers is zero, to mark its end.

Example of usage:

```
% cat a10/split1.c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

char **split(char *s);

int main()
{
    char **p = split("3.red5.green");

    for (int i = 0; i < 3; i++) {
        char *s = p[i];
        printf("p[%d] = '%s'\n", i, p[i] ? p[i] : "<null>");
        if (s)
            free(s);
        }

    free(p);
}
```

You'll probably want to use `getnth` from problem 2. To avoid "coupling" these two problems, the tester uses my `getnth` solution, in object-file form, in `a10/getnth.o`. Here's a `gcc` invocation that does the same thing the tester does:

```
gcc -o split1 a10/split1.c split.c a10/getnth.o
```

Execution:

```
% split1
p[0] = 'red'
p[1] = 'green'
p[2] = '<null>'
```

If `split` is called with an empty string, it returns a pointer to a dynamically allocated one-element array; the value of the element is zero.

`split` should make no about assumptions the length of the string being split or the number of strings it is being split into. If sufficient memory is available via `malloc` to hold the result, then `split` should be able to handle it.

Problem 4. (5 points) `build.c`

Write a C function `char *build(char **strings)` that creates and returns a dynamically allocated `getnth`-style string from `strings`, a zero-terminated array of pointers to strings.

Here is a test program:

```
% cat a10/build1.c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
char *build(char **strings);
int main()
{
    char *s1[] = { "a", "test", "of", "build", 0 };
    char *s2[] = { "1", "", "22", "", "333", 0 };

    char **arrays[] = { s1, s2, 0 };
    char ***next;

    for (next = arrays; *next; next++) {
        char *s = build(*next);
        puts(s);
        free(s);
        }
 }

% gcc -o build1 a10/build1.c build.c
% build1
1.a4.test2.of5.build
1.10.2.220.3.333
```

`build` should make no assumptions about the number of strings it is given or the total length of the string being built. If sufficient memory is available via `malloc` to hold the result, then `build` should be able to handle it.

Implementation note: Use `sprintf` to convert lengths into character strings.

Problem 5. (15 points) picklines.c

For this problem you are to write a C program `picklines` that reads lines on standard input and prints lines according to one or more command-line specifications. Here's a simple file to work with:

```
% cat a10/picklines.1
a
b
c
d
e
f
```

Let's use `picklines` to print the third, first, and last line of the file:

```
% picklines 3 1 -1 < a10/picklines.1
c
a
f
```

As you can see, line numbers are one-based. You can also see that reverse numbering is supported: `-1` specifies the last line.

A specification can be a range of lines:

```
% picklines 2:4 6:1 < a10/picklines.1
b
c
d
f
e
d
c
b
a
```

Note that `6:1` produces reversed output: line 6, then line 5, etc., through line 1.

Negative numbers can be used in a range, too:

```
% picklines -1:1 < a10/picklines.1
f
e
d
c
b
a
% picklines -3:-2 < a10/picklines.1
d
e
```

If a specification contains a line that is out of range, that specification is silently ignored. Here's a series of invalid specifications followed by a single valid specification, which does produce a line of output:

```
% picklines 0 7 -7 1:10 -7:1 3 < a10/picklines.1
c
```

Note that zero is never a valid line number.

Assume specifications are well formed—either a single integer or two integers separated by a colon. You won't see things like `1:`, `--3`, or `3::x`.

Assume that input lines are less then 1000 characters in length. There may be any number of specifications on the command line. There may be as many as 9,223,372,036,854,775,807 (`LONG_MAX`) input lines, so use a `long` to represent line numbers.

Implementation notes

To keep things simple, I recommend that you read all lines into memory. Use the technique shown with realloc2.c (slide 382) to create an array of char pointers that's expanded as needed.

On rectangle.c on assignment 6 you used scanf to read numbers from standard input. A related function is sscanf (note the double 's'), which "reads" from a string. I hope to spend some time on scanf and sscanf in lecture before the semester is done but for now let me just show an example of using sscanf to parse line specifications. Here is a10/parsespec.c:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    long first, last;
    int nums = sscanf(argv[1], "%ld:%ld", &first, &last);

    printf("nums = %d, first = %ld, last = %ld\n",
        nums, first, last);
}
```

Here are two interactions with it:

```
% parsespec 1:100
nums = 2, first = 1, last = 100

% parsespec -5
nums = 1, first = -5, last = 140725490698960
```

Note that the return value implies whether the specification is of the form *N* or *N:M*.

You can experiment with the executable, a10/parsespec.

Problem 6. (TODO points) alloc.c

In this problem you are to implement a portion of a memory allocator. An industrial strength memory allocator must be fast and memory-efficient. This allocator is neither—it wastes a lot of space and is relatively slow. However, it has a very simple design and also provides detection of leaks, and frees of non-allocated blocks. And, writing a memory allocator will give us additional insight about using memory allocation.

You need to write relatively little code for this problem; the harder part of the problem is understanding the design of this simple allocator. Once you "get it", I think you'll find the coding to be easy, but you may need to read through this write-up several times to "get it".

We'll do more with this allocator on assignment 11, so time spent now will be of benefit then, too.

A first step is to be sure you understand the idea of the trivial allocator on slide 372: It first sets aside a big block of memory as an array named memory_pool. next_pool_addr is pointed at the first byte of memory_pool. If malloc(1000) is called, next_pool_addr is returned, and then advanced 1000. A following call to malloc(50) would return the current value of next_pool_addr and then do next_pool_addr += 50. No record is kept of allocations, and free does nothing, but as long as less than 100,000,000 bytes are needed, it'll work just fine.

Here is the interface for the allocator that's the focus of this problem: (`a10/alloc.h`)

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

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

void show_pool(char *label);
```

I'll be giving you the code for `alloc_block`. **Your task on this problem is to write the other two routines: `free_block` and `show_pool`.**

A key simplification in this allocator is the use of fixed-size blocks. Every allocation consumes `BLOCK_SIZE` bytes of the memory pool, and no allocation can be more than `BLOCK_SIZE` bytes. A further simplification is that there can only be a fixed-number of active allocations. Those two simplifications lead to this declaration:

```
char mem_pool[BLOCK_SIZE*MAX_ACTIVE_ALLOCS];
```

The above is a line of code that will appear in your version of `alloc.c`. `mem_pool` is a global variable—the declaration is NOT placed inside a function.

Even though each allocation consumes `BLOCK_SIZE` bytes, another array, `block_sizes`, also a global, tracks the amount of memory that was requested via `alloc_block`. Here is its declaration:

```
int block_sizes[MAX_ACTIVE_ALLOCS];
```

With those two pieces of the implementation in hand, let's consider some code that makes use of the allocator. Here is a loop that allocates 3 blocks:

```
char *a[10];

for (int i = 1; i <= 3; i++) {
    a[i-1] = alloc_block(i*10, "loop 1");
```

`alloc_block` simply examines each element of `block_sizes` in turn until it finds an element (`N`) that is zero. It sets `block_sizes[N]` to the number of bytes requested and returns the address of `mem_pool[BLOCK_SIZE*N]`.

After the above loop is done, `a[0]` is `&mem_pool[BLOCK_SIZE*0]`, `a[1]` is `&mem_pool[BLOCK_SIZE*1]`, and `a[2]` is `&mem_pool[BLOCK_SIZE*2]`.

Further, `block_sizes[0]` is `10`, `block_sizes[1]` is `20`, and `block_sizes[2]` is `30`. Because `block_sizes` is a global, all elements are set to zero when the program begins execution. The other elements of `block_sizes` will thus be zero.

The second argument to `alloc_block` is a *tag*. The tag provides a way to associate an allocated block with a particular call to `alloc_block`. In the case above the programmer has chosen `"loop 1"` as the tag.

`block_tags`, another array in `alloc.c`, "parallels" `block_sizes`:

```
      char *block_tags[MAX_ACTIVE_ALLOCS];
```

`block_tags[i]` holds the tag associated with `block_sizes[i]`. Continuing with the series of `alloc_block` calls in the example, `block_tags[0]`, `block_tags[1]`, and `block_tags[2]` each reference the string `"loop 1"`.

Next, a block is freed:

```
      free_block(a[2], "a");
```

As mentioned above, the value held in `a[2]` is `&mem_pool[BLOCK_SIZE*2]`—the third block in the pool. <u>Because of the use of fixed size blocks</u>, `free_block` can easily calculate which block is being freed:

```
      block_num = (addr - mem_pool) / BLOCK_SIZE;
```

In this case, `block_num` will be `2`.

To indicate that the block is now available for subsequent allocation, its entry in `block_sizes` is zeroed:

```
      block_sizes[block_num] = 0;
```

Next, another block is allocated:

```
      a[5] = alloc_block(200, "next");
```

`alloc_block` will scan `block_sizes` and find that `block_sizes[2]` is zero. It will set `block_sizes[2]` to `200` and return `&mem_pool[BLOCK_SIZE*2]`.

At any point in time, the total amount of memory currently allocated can be calculated by summing the values in `block_sizes[0]` through `block_sizes[MAX_ACTIVE_ALLOCS-1]`.

In addition to the code above, I am giving you the code for `alloc_block`. Here it is:

```
      void *alloc_block(int nbytes, char *tag)
      {
          if (nbytes <= 0 || nbytes > BLOCK_SIZE)
              return 0;
          for (int i = 0; i < MAX_ACTIVE_ALLOCS; i++) {
              if (block_sizes[i] == 0) {
                  block_sizes[i] = nbytes;
                  block_tags[i] = tag;

                  char *block_start = &mem_pool[i*BLOCK_SIZE];
                  return block_start;
```

```
            }
        }
    return 0;
}
```

As described above, `alloc_block` first searches for an empty block, indicated by `block_sizes[i]` being zero. If found, it sets the size and tag, and returns the address of the block. If no blocks are available, it returns 0.

The above code can be found in `a10/alloc_starter.c`. Copy that file to `alloc.c` in your directory and start with it.

<u>Recall: Your task on this problem is to write two routines: `show_pool` and `free_block`.</u>

The `void show_pool(char *label)` function prints a report that shows the status of the pool. The first line of the report is the `label` passed to `show_pool`. For each active block it then prints the block number (in the range `0` to `MAX_ACTIVE_ALLOCS`), the amount of memory allocated by the user for the block, the address of the block, and the tag associated with the block. Finally, it prints the number of allocated blocks and the total amount of memory allocated.

The following program, `a10/alloc1.c`, incorporates the examples above and interleaves two calls to `show_pool`:

```
#include <string.h>
#include <stdio.h>

#include "/cs/www/classes/cs352/fall15/a10/alloc.h"

int main()
{
    char *a[10];

    for (int i = 0; i < 3; i++)
        a[i] = alloc_block((i+1)*10, "loop 1");

    show_pool("After loop:");

    free_block(a[2], "a");

    a[3] = alloc_block(200, "next");

    show_pool("Ready to exit:");
}
```

Compilation and execution:

```
% gcc alloc.c a10/alloc1.c
% a.out
After loop:
Block 0: 10 bytes at 0x61e520, tag: "loop 1"
Block 1: 20 bytes at 0x61e920, tag: "loop 1"
Block 2: 30 bytes at 0x61ed20, tag: "loop 1"
Total: 3 allocated blocks, 60 allocated bytes
Ready to exit:
Block 0: 10 bytes at 0x61e520, tag: "loop 1"
Block 1: 20 bytes at 0x61e920, tag: "loop 1"
Block 2: 200 bytes at 0x61ed20, tag: "next"
Total: 3 allocated blocks, 230 allocated bytes
```

Next, here's what `void free_block(void *addr, char *tag)` needs to do:

(1) Be sure that `addr` is an address returned by `alloc_block`.

(2) Be sure that the block is currently allocated.

(3) Zero the corresponding entry in `block_sizes`.

Here is a program, `a10/alloc2.c`, that has two memory management errors that `alloc.c` detects.

```
#include <string.h>
#include <stdio.h>

#include "/cs/www/classes/cs352/fall15/a10/alloc.h"

int main()
{
    char *p1 = alloc_block(100, "p1");
    char *p2 = alloc_block(200, "p2");
    *p2++ = 'x';

    free_block(p1, "A");
    free_block(p1, "B"); // duplicate free

    free_block(p2, "C"); // free of advanced pointer

    show_pool("Done!");
}
```

Output:

```
free_block(0x61e520, B): free of non-allocated block
free_block(0x61e921, C): bad address
Done!
Block 1: 200 bytes at 0x61e920, tag: "p2"
Total: 1 allocated blocks, 200 allocated bytes
```

If an error is detected, `free_block` simply prints a message, which should be in the exact format shown above, and exits. Here are the two `printf` format strings used:

```
"free_block(%p, %s): bad address\n"
"free_block(%p, %s): free of non-allocated block\n"
```

On many slides you've seen pointers printed using a "`%lu`" but note that "`%p`" is the correct, portable way to print a pointer value. The C11 standard describes `%p` like this: *"The argument shall be a pointer to* `void`*. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner."*

Problem 7. 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 `a10/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 `a10/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.

`a10/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 < a10/delivs)
  81  196 1395 warmup.c
  34   81  627 getnth.c
  24   54  374 split.c
  29   71  575 build.c
  84  253 1903 picklines.c
  68  184 1564 alloc.c
```

```
 320  839 6438 total
% for i in $(grep -v txt a10/delivs); do echo $i: $(tr -dc \; <
$i | wc -c); done
warmup.c: 29
getnth.c: 14
split.c: 12
build.c: 14
picklines.c: 40
alloc.c: 27
```

There are few comments in my code.

Miscellaneous

This assignment is based on the material on C slides 1-397.

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, and has completed the previous assignments, and has done the required reading will need 8-10 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 eight 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 five 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.)