CSC 4730 Fall 2023 Project 5

In this project you will write programs that implement three free space allocator algorithms.

Objectives

The educational purposes of this project include:

- You have a concrete opportunity to implement two programs with massive code reuse and with good design, inheritance.
- You will master free space management algorithms that are in use today to manage a process's heap space.
- You will design and implement units tests according to documentation.

Deadlines

You and your assigned partner have 7 days to complete this assignment plus one grace day. If you have not turned anything in by 11:59 PM on the eighth day, you will win a grade of zero. Therefore, to receive partial credit, hand in something before the project expires!

Overview

These will implement:

1. Slab
2. Next Fit
3. Best Fit

You can choose to implement these in C or C++. I would definitely choose C++ - see here.

You will implement these as three separate executables but you may share code between them if you are able to design some common data structures and algorithms.

Next Fit and Best Fit read a list of commands from standard input and write results to standard output. Slab executes a prescribed sequence of steps producing output but taking no input.

You have **NO FREEDOM WHATSOEVER** as to the format of the output as your programs will be tested by shell script (provided to you with some but not all test data).

Environment

You must use WSL on Windows or the Terminal on the Mac.

Slab Allocator

This program is *not* interactive. You must write the tests described below, in the sequence described. That is, you will hard code the entire described sequence.

The slab allocator will work on slabs of 512 bytes. A "gulp" is when the allocator has run out of slabs to dole out and needs to allocate a large group of slabs all at once. The gulp size is 128. That is, when the allocator needs to allocate more slabs because it has run out, the allocator will make a gulp of 128 * 512 bytes.

Slabs which are returned to the pool, are pushed onto a stack. If slab *a* is freed, the next slab to be allocated (without intervening frees) will be *a*.

Your program **must** follow these steps which will exercise your safeguards and error checking. Any tests that fail must print the error messages given to you below and must cause the program to return 1.

If no errors are found, the program must return 0.

Note: In some of these steps, it is an error to have NOT found an error. In other words, you're supposed to trigger error checking code you must write and if the error checking code does not trigger, *that's* the real error that must be reported and your program exit with a return code of 1.

Steps to Hard Code

1. Allocate 256 slabs consecutively.

   This will cause two gulps and 256 print outs from your allocator.

   You will test to ensure the number of available slabs is exactly 0 because step 1 allocates the exact right number of slabs to empty all available.

2. All 256 slabs are then freed. The order in which they are freed is not important.

   You will test to ensure that the resulting number of available slabs is exactly 256. If the number of available slabs is not 256 you will print the error given to you below and exit the program with return code 1.

3. One slab will be allocated and then freed. Remember its address.

4. Another slab will be allocated and then freed. Remember its address.

5. The two addresses are compared - they should be the same and reported so.

   If the test fails, your program must exit with a return code of 1.

6. You will attempt to free a slab giving a null address. This should be flagged as an error. If you don't catch the error, the message given below must be printed and your program exit with return code 1.

7. You will attempt to free too many buffers. At this time, all buffers should have been returned to the free pool. Repeat an attempt to free the buffer you remembered in Step 4. Because the number of available buffers reads full, the "too many frees" error should be triggered.

   NOTE THAT YOU SHOULD CHECK FOR TOO MANY FREES BEFORE CHECKING FOR A DOUBLE FREE.

8. Now allocate TWO slabs. Return the second one TWICE. This should trigger a double free error. If the error is not found, report the failure to catch the error using the text given below and exit your program with a return code of 1.

9. You will attempt to free a make believe slab whose address is 12345. This will, of course, be flagged as an error. If you do not catch the error, you must note this using the text given below. Then your program must exit with a return code of 1.

Here is the output you must produce LETTER FOR LETTER... it is long.

Line 1 says:

```
Gulping
```

This is the output you must emit when you gulp space for 128 512 byte slabs.

Here are the next three lines. As you can see, slabs are being allocated as per step 2 above. Notice the number at the end of each line. This number is the count of the available slabs left to allocate. One slab was given away, hence the number 127. Another is given away, hence the number 126. Etc.

```
Allocating Slab  127
Allocating Slab  126
Allocating Slab  125
```

The following is key - found on Lines 128 to 131:

```
Allocating Slab    1
Allocating Slab    0
Gulping
Allocating Slab  127
```

This means you correctly gulped again when you ran out of available buffers. The next 128 lines are allocating the newly gulped buffers.

Line 259 is printed after you have freed all of the slabs you allocated. If this test fails, you must print:

```
"Number of Available Slabs should be 0. Is: <some number> (Wrong)
```

Then your program must exit with return code of 1.

Line 260 caused by step 2 above.

Line 261 is the test called for in step 2 above. If the test fails you must print:

`Number of Available Slabs should be 256. Is: <some number> (Wrong)`

Then your program must exit with a return code of 1.

You are now working on Step 3, 4 and 5 above.

Assuming you're still running, you must allocate one slab. Remember its address. Free it. This accounts for line 262 and is Step 3.

Allocate a second slab. This accounts for line 263 and is Step 4. Remember its address.

You must test that the addresses returned by both of the last allocations are **the same**. This is Step 5 above.

THIS MEANS YOUR FREE LIST IS MANAGED AS A LIFO - LAST IN - FIRST OUT.

Line 264 is printed if the results match. If the two values are NOT the same, this is the error you must print:

`Alloc / Free / Alloc Test " failed`

Then your program exits with a return code of 1.

Assuming you are still running you are proceeding to Step 6. You must test your handling of attempting to free a nullptr.

Line 265 shows success. If you fail to catch the attempt to free a nullptr, you must print:

`Did Not Catch Attempt to Free NULL`

If the test failed, you must then exit with a return code of 1.

Assuming you are still running you are moving on to Step 7. At this time, all buffers should have been returned to the free pool. Repeat an attempt to free the buffer you remembered in Step 4. Because the number of available buffers reads full, the "too many frees" error should be triggered.

NOTE THAT YOU SHOULD CHECK FOR TOO MANY FREES BEFORE CHECKING FOR A DOUBLE FREE.

Line 266 shows the correct result. If your test failed, you must print:

`Did Not Catch Freeing of Too Many Buffers`

and exit your program with a return code of 1.

Now you are on to Step 8.

Line 267 shows an allocation.

Line 268 shows an allocation. Free this slab twice. This should trigger line 269. If it does not, you must print:

```
Did Not Catch Attempt to Double Free
```

and exit your program with a return code of 1.

Assuming you're still running, now you are on Step 9.

You are to attempt to free a slab at address 12345. This of course is nonsense - you must catch this error. If you correctly do, print line 270. If you don't you must print:

```
Did Not Catch Attempt to Free BAD Address
```

Finally, print line 271 and exit your program with return code of 0.

The destructors will run which should **FREE EVERY BYTE YOU HAVE ALLOCATED WHICH IS EASY TO DO SINCE YOU REMEMBERED THE BASE ADDRESS OF EVERY GULP, RIGHT?**

Setting Expectations

My implementation with all error checking and some comments and blank lines ran 207 lines. This is just to set your expectations and is not a challenge. My program, by the way, is shorter that the text describing it.

Next Fit and Best Fit

Yes, TWO MORE programs to implement in this project. Yikes.

If you DESIGN your implementation right, both programs will be essentially identical.

My implementations differ in length by 3 lines and sixty seven bytes. Of course, this counts only length. There are more than three different lines and sixty seven different characters. For example, one difference would be:

```
BestFit nf(size_of_ram);
```

versus

```
NextFit nf(size_of_ram);
```

To repeat, if you design this right, most of the code will be shared between the two programs.

NOT REAL MEMORY

This is a simulation. All sizes are given in kibibytes. No actual calls to `malloc()` are made in these programs. You are *making believe* you're managing free user

memory. Hint: you might create instances of classes that might get pushed onto and removed from various `vector`s.

Command Line Arguments You Must Support

All three executables must implement the following command line options:

Option	Argument	Meaning	Default
-h	none	prints this help text	none
-k	number of KiB	sets the maximal size of free memory	512

Interactive Commands You Must Implement

Command	Argument	Meaning
q	None	Exits the program
p	None	Prints the Free and Allocated lists
a	size	Attempts to Allocate `size` kibibytes
f	addr	Attempts to Free an allocation starting at `addr`
#	text	A `#` means skip the entire line

Test Files

A lot of tests are provided to you. These are what will be used to grade your work. You are also provides with a shell script that will run the tests *and tell you if your program is correct!*.

Remember, you must match output letter for letter to pass.

With that said, I have programmed the `diff` command to be somewhat permissive.

Here is an example of using one of the tests:

```
./expected_output_test.bash -a ../best -i test_20
```

Notice this assumes you have changed directories to a subfolder in relation to where your executables live.

Tests numbered 19 or less are for Next Fit.

Tests numbered 20 and more are for Best Fit.

Here is the output from `test_20` using the test script.

```
Test input file:       test_20.txt
Expected output file:  test_20.expected_output.txt
Expected output (must match letter for letter):
Free List
```

```
 Index   Start  Length
[0000]       0     512

Allocated List is empty

Your output:
Free List
 Index   Start  Length
[0000]       0     512

Allocated List is empty

Differences:
PASSED
Test finished
```

Notice there are no prompts (see next) and that both your output and the expected output are given. If there were any differences, the differences would have been printed as well.

Disappearing Prompt

You must detect if you are taking input directly from an interactive terminal or not. Hint: `isatty()`. If you are taking input directly from the console, you must print a `>` plus a space as the interactive prompt.

If you detect that you are reading input from something *other* than a terminal, skip printing the prompt entirely.

Suggestion

Leverage your C++ knowledge to make implementation easier. You can use `vector` and `algorithm` for example. Mentioning `algorithm` might be a hint.

Next Fit

Beginning at the beginning literally, First Fit starts looking for free memory beginning with the first free section. This is trivial to implement but leads to lots of fragmentation. An improvement is Next Fit.

Next Fit remembers where it stopped checking for free sections the last time. For example, if the 10th free spot was chosen to be split, and not used up completely, there would still be a (smaller) 10th free spot. THIS is where Next Fit would start looking for the next allocation.

Clearly, you'll need to handle a number of corner / special cases. Here are two - there might be more:

- What happens when you think you'll start looking at what was the last free section but that section was used completely so it no longer exists?
- When happens when you reach the end of the list of free sections having started searching in the middle of the list?

Next Fit Tests

Test Number	Purpose
01	Tests proper initialization
02	Tests giving away all of memory
03	Tests giving away all of memory and then taking it back
04	Tests coalescing
05	Tests coalescing
06	Tests Coalescing
07	Tests something
08	Tests making a hole then giving part of it away
09	Tests allocation too big for hole
10	Test the NEXT fit behavior
11	Test the NEXT fit behavior with a failure
12	Tests next index wrap around
13	Tests bad free
14	Tests bad free
15	Tests bad free

Best Fit

Best fit differs from Next Fit in that it will match the current allocation request to the "best" free section i.e. it will allocate from the smallest section that is big enough to handle the request.

I smell a `sort`. Maybe more than one? Depends on your implementation. I used two different sorts but maybe that's just me. Your mileage may vary.

Best Fit Tests

Test Number	Purpose
20	Tests initialization
21	Tests giving away all of memory.
22	Tests giving away all of memory and then taking it back.
23	Test Free From Middle - No Coalesce
24	Test Free From Beginning - No Coalesce
25	Test Free From End - No Coalesce
26	Test Free With Coalesce Forwards
27	Test Free With Coalesce Backwards

Test Number	Purpose
28	Test Free With Coalesce Forwards and Backwards
29	Tests passing over bigger block for smaller one
31	Tests bad free
32	Tests bad free

Provided makefile

If you create main source code files named slap.cpp, next.cpp and best.cpp you could be able to leverage the included makefile. Then it is easy to build your programs like so:

```
$ make slab
```

What to Hand In

Zip your makefile and all of your .c and / or .cpp source files and any header files into a single zip file containing NO sub-folders. If your zip file contains a subfolder, I will remove 5 points.

Partners

- You must use only the partner I assign you.
- Only 1 person should hand in code. The code should clearly state who the partners are in a comment at the top of the main source file.
- The non-code-submitting partner must submit a text file "partner.txt" that states who the partners are.
- Failure to list partners correctly as described above removes 5 points from your grade.

Grading

The class will receive a rubric when grading is complete that describes errors and penalties.

Both partners get the same grade without exception.