CS107 March 17th, 2025
Winter 2025

# CS107 Final Examination

This is a closed book, closed note, closed computer exam, though you're permitted to refer to the reference sheet I've provided.

You have 180 minutes to complete all problems. You don't need to **`#include`** any libraries, and you needn't use **`assert`** to guard against any errors. Understand that most points are awarded for concepts taught in CS107, and not prior classes. You don't get many points for **`for`**-loop syntax, but you certainly get points for proper use of **`&`**, **`*`**, and the low-level C functions introduced in the course. If you're taking the exam remotely and have questions, you can text or telephone Jerry at 415-205-2242.

Good luck!

SUNet ID (@stanford.edu): _____

Full Name: _____

I accept the letter and spirit of Stanford's Honor Code.

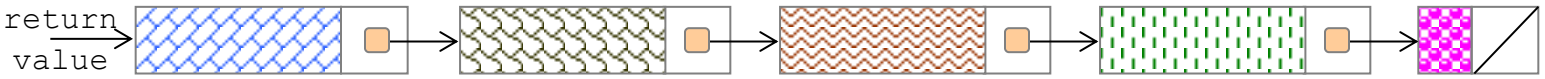
[signed] _____

| | | Score | Grader |
|---|---|---|---|
| 1. Chunks | [10] | _____ | _____ |
| 2. Generics and Function Pointers | [10] | _____ | _____ |
| 3. x86-64 and **`gcc`** optimizations | [20] | _____ | _____ |
| 4. Runtime Stack | [10] | _____ | _____ |
| 5. Custom Allocators | [20] | _____ | _____ |
| **Total** | **[70]** | _____ | _____ |

**Problem 1: Chunks [10 points]**

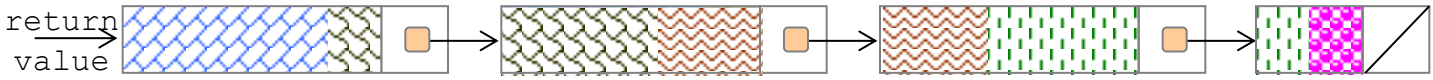
Given the address of some binary stream, its size in bytes, and a chunk size in bytes, write a function called **`chunkify`** that creates a **heap-based** linked list of data chunks. So, given the address of the following, 1700-byte data stream:

a call to **`chunkify(stream, 1700, 400)`** would return the address of a **`NULL`**-terminated list that looks like so:




Most if not all nodes are a **`400`**-byte portion—or rather, chunk—of the original stream followed by the address of the next node in the list. The final node might be smaller than the others, and in this case, it is, because it only needs to store the last 100 bytes of the full data stream. Of course, the accumulation of all chunks should match the byte sequence of the original stream.

A call to **`chunkify(stream, 1700, 500)`** would return a different list:




This time, each node houses **`500`**-byte chunks, save for the last, which is **`200`** bytes instead. This whole process is akin to the process that content servers at, say, Netflix and Spotify use to stream audio and video to web browsers and mobile devices.

Use the next page to present your implementation of **`chunkify`**. Note that, because node sizes can't be determined at compile time, there's no universal **`struct`** definition to use, so your implementation should be implemented without one.

```
void *chunkify(void *stream, size_t length, size_t chunk) {
```

**Problem 2: Generics and Function Pointers [10 points]**

Write a generic **`remove_repeats`** function that accepts an array and replace all runs of identical elements with a single instance. For instance, given the following **`int`** array:

| 18 | 65 | 65 | 65 | 65 | 29 | 65 | 40 | 40 | 29 | 89 | 89 | 51 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

the function would remove neighboring duplicates—in some cases, many of them, as with the 65's—and would update the array in place to look like this:

| 18 | 65 | 29 | 65 | 40 | 29 | 89 | 51 |
|---|---|---|---|---|---|---|---|

Note that the last of the five 65s is not removed, since it's separate from the group of four that precedes it. And because the updated array length is otherwise unknown to the client, **`remove_repeats`** needs to return the new effective length. So, in the case of the above example, the supplied length would be 13 but the return value would be 8. Of course, we want our implementation to be fully generic! That's possible provided we supply the element size and an appropriate comparison function in addition to the array's base address and its length.

a) [5 points] In the space below, complete the partial implementation of **`remove_repeats`**. Recall the return value of the comparison function signals just how different the two items being compared really are, but a return value of 0 means the two elements being compared are identical as far as **`remove_repeats`** and its comparison function are concerned.

Your implementation is **required** to traverse from the end of the array to the front, and the **`len`** parameter should be decremented accordingly so that it can serve as the return value.

```
size_t remove_repeats(void *base, size_t len, size_t elem_size,
                      int (*cmp_fn)(void *, void *)) {

   for (ssize_t i = len - 2; i >= 0; i--) {


      void *first = ____________________________________;


      void *second = (_____________) first + _________________;
      // complete the rest of the loop body









   }

   return len;
}
```

b) [3 points] Assume now that you need to compress runs of C strings, where neighboring strings are considered the same if they simply **begin with the same letter**. Consider the following:

```
void test_strings() {
   char *strings[] = {
      "accrue", "accrue", "ace", "beautiful", "burgeoning",
      "clockwork", "clockwork", "clockwork", "clockwork", "colorful",
      "dahlia", "deride", "excellence", "excellence", "fanatic"
   };
   int len = sizeof(strings)/sizeof(strings[0]);
   print_string_array("Before:", strings, len);
   len = remove_repeats(strings, len, sizeof(char *), lead_char_cmp);
   print_string_array(" After:", strings, len);
}
```

```
Before: accrue accrue ace output omitted for brevity excellence excellence fanatic
 After: accrue beautiful clockwork dahlia excellence fanatic
```

Assuming an obvious implementation of **`print_string_array`**, the only missing details come with the implementation of **`lead_char_cmp`**, which compare C strings by just their leading characters (and nothing else). Present your implementation of **`lead_char_cmp`**.

```
int lead_char_cmp(void *one, void *two) {
   // your implementation should be at most three lines




}
```

c) [2 points] Consider the following implementation of **another** generic comparison function, which interprets the first address as a **`char *`** but the second as a **`char **`**, as with:

```
int string_compare(void *one, void *two) {
   char *s1 = one;
   char *s2 = *(char **)two;
   return strcmp(s1, s2);
}
```

Explain why **`string_compare`** can't possibly be used to **`qsort`** an arbitrary array of C strings, but it could technically be used to **`bsearch`** a sorted array of C strings for a specific key.

**Problem 3: x86-64 and gcc Optimizations [20 points]**

The assembly code presented on the lower right was generated by compiling the **`labrador`** function without optimization using **`-O0`**.

a) [14 points] First, fill in the blanks below so that **`labrador`** is programmatically consistent with the unoptimized assembly you see on the right. Note that the C code is nonsense and should just be a faithful reverse compilation. You may not typecast anything. Remember that **`size_t`** and all pointers are 8 bytes in size.

```
1129: sub    $0x38,%rsp
112d: mov    %rdi,0x18(%rsp)
1132: mov    %rsi,0x10(%rsp)
1137: mov    %rdx,0x8(%rsp)
113c: mov    0x10(%rsp),%rax
1141: add    $0x50,%rax
1145: mov    (%rax),%rax
1148: sub    0x18(%rsp),%rax
114d: mov    %rax,0x28(%rsp)
1152: mov    0x18(%rsp),%rax
1157: and    $0xf,%eax
115a: test   %rax,%rax
115d: jne    11b0 <labrador+0x87>
115f: mov    0x28(%rsp),%rax
1164: lea    0x0(,%rax,8),%rdx
116c: mov    0x10(%rsp),%rax
1171: lea    (%rdx,%rax,1),%rcx
1175: mov    0x8(%rsp),%rdx
117a: mov    0x28(%rsp),%rax
117f: mov    %rcx,%rsi
1182: mov    %rax,%rdi
1185: callq  1129 <labrador>
118a: mov    0x10(%rsp),%rdx
118f: mov    (%rdx),%rdx
1192: add    %rax,%rdx
1195: mov    0x10(%rsp),%rax
119a: mov    %rdx,(%rax)
119d: mov    0x28(%rsp),%rdx
11a2: mov    %rdx,%rax
11a5: add    %rax,%rax
11a8: add    %rdx,%rax
11ab: mov    %rax,0x18(%rsp)
11b0: mov    0x18(%rsp),%rax
11b5: add    $0x38,%rsp
11b9: retq
```

```
size_t labrador(size_t beagle, size_t *poodle, char *boston) {

    size_t boxer = ____________________________________________________;

    if (_______________________________________________________) {

        _______________ += labrador(_____________, _____________, _____________);

        beagle = ____________________________________________________;
    }

    return ____________________________________________________;

}
```

Now, study the aggressively optimized version of **labrador** on the right and answer the questions below.

```
1150: mov    %rdi,%rax
1153: test   $0xf,%al
1155: je     1160 <labrador+0x10>
1157: retq
1160: push   %rbp
1161: push   %rbx
1162: mov    %rsi,%rbx
1165: sub    $0x8,%rsp
1169: mov    0x50(%rsi),%rbp
116d: sub    %rdi,%rbp
1170: lea    (%rsi,%rbp,8),%rsi
1174: mov    %rbp,%rdi
1177: callq  1150 <labrador>
117c: add    %rax,(%rbx)
117f: add    $0x8,%rsp
1183: lea    0x0(%rbp,%rbp,2),%rax
1188: pop    %rbx
1189: pop    %rbp
118a: retq
```

b) [2 points] The second of the two lines under jurisdiction of the **if** test — specifically, the one that reassigns **beagle** — compiles to just one assembly code instruction. Which one is it?

c) [2 points] Our **labrador** function clearly takes three arguments, yet only **%rdi** and **%rsi** are updated just prior to the recursive call at address **1177**. Why isn't **%rdx** updated?

d) [2 points] Recall that **code motion** is an optimization technique where C statements are reordered or even moved with hopes of reducing the dynamic instruction count. Identify how code motion is being used here and why the compiler knows it's okay to use it.

**Problem 4: Runtime Stack [10 points]**

You're working as a security engineer for a fintech company that takes its cybersecurity very seriously. You've been alerted by some security researchers that they're able to gain root access to information only those working at the company should have access to. You know which function to scrutinize, and that function is right here:

```
bool login_as_root() {
   struct {
      char supplied[16];
      size_t canary;
      char secret[16];
   } data;

   data.canary = 0; // set to 0 and confirm it remains 0 before returning true
   load_secret_password(data.secret, sizeof(data.secret));
   for (size_t i = 0; i < 10; i++) { // up to 10 attempts
      printf("Enter root password: ");
      gets(data.supplied); // populates data.supplied without bounds checking
      if (strcmp(data.supplied, data.secret) == 0) { return data.canary == 0; }
      printf("Incorrect. Try again.\n");
   }

   return false;
}
```

You don't question **`load_secret_password`**, because that's verified to be correct on a daily basis. You suspect the **`gets`** function is the problem, **since it populates the supplied buffer with whatever the user types into standard input, without bounds checking**. However, you still want to produce a sequence of 10 or fewer passwords that prompt the above to return **`true`** regardless of what the true secret root password is. So, you fire up **`gdb`** to snoop around and analyze **`login_as_root`**'s stack frame. Here's what you find.

```
Breakpoint 1, login_as_root () at canary.c:73
13 bool login_as_root() {
(gdb) p/x &data.supplied
$4 = 0x7fffffffe690
(gdb) p/x &data.canary
$5 = 0x7fffffffe6a0
(gdb) p/x &data.secret
$6 = 0x7fffffffe6a8
```

You confirm what you already suspected: the 16-byte **`secret`** sits atop the 8-byte **`canary`** sits atop the 16-byte **`supplied`**, with no intervening padding whatsoever. **`gets`** is obviously being exploited to fool the function into returning **`true`**. But how?

a) [3 points] By examining the C code and the **gdb** output, give an example of a first password that will fully overwrite **supplied**, fully overwrite **canary**, and update **secret** to be whatever your official SUNet ID is (e.g., mine is **poohbear**). You don't expect **supplied** and **secret** to match, so you needn't care just yet how **supplied** and **canary** are updated.

b) [4 points] Describe how the next several passwords can be leveraged to restore **canary** to the 0 it ultimately needs to be again before populating **supplied** with one final password that matches **secret**. What sequence of passwords can you use to ensure **canary** eventually gets set back to 0? And what is the one final password that coerces the function to return **true**?

This same fintech company has more than a few bugs in its platform. In addition to the authentication issue you just worked through, you hear of the occasional server getting stuck in what seems like an infinite loop, thereby requiring a server reboot.

You've isolated the issue to a function that loads an array of stock share counts—one for each of 40 stocks in a stock fund—and sells 5 shares of each. The problem can be reproduced via a simplified version of the code presented here:

```
void update_portfolio() {
   long stocks[40];
   read_portfolio(stocks, 40); // this first function works fine
   adjust_portfolio(stocks, 40, -5);
}
```

You fully trust the **`read_portfolio`** to properly read in the share counts for each of the 40 stocks from some database, but you notice the implementation of **`adjust_portfolio`** has an off-by-one error:

```
void adjust_portfolio(long scores[], size_t n, long delta) {
   for (size_t i = 0; i <= n; i++) scores[i] += delta;
}
```

Clearly the **`<=`** should be a **`<`**, and you'll obviously fix it. But you wonder why that would cause the server get stuck and never advance beyond the call to **`update_portfolio`**. You to look at the assembly for the **`update_portfolio`** call, and you encounter this:

```
121e: e8 b3 ff ff ff          callq  11d6 <update_portfolio>
1223: 48 83 c4 10             add    $0x10,%rsp
```

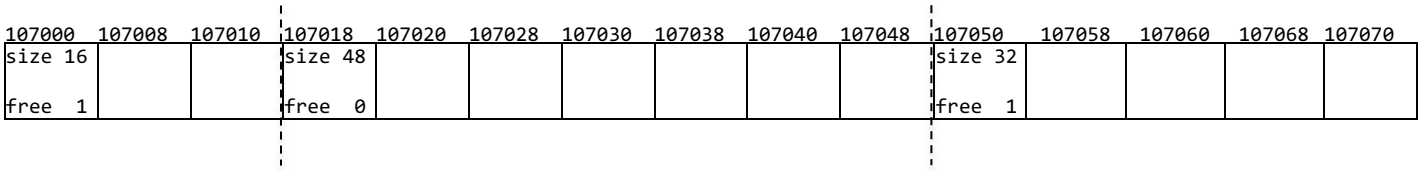
You look at the first line of assembly in **`adjust_portfolio`** and confirm its stack frame grows from 0 bytes to 0x140 bytes—or rather, 320 bytes for 40 longs, which is just enough space for the 40 share counts. Restated, there's no padding anywhere in the stack frame.

```
00000000000011d6 <update_portfolio>:
   11d6:  48 81 ec 40 01 00 00    sub    $0x140,%rsp
```

c) [3 points] **`%rsp`** through **`%rsp + 320`** house the 40 stock share counts, but the eight bytes above the array house something else. What is directly above the array in the stack segment, and why does decrementing it by 5 cause the program to seemingly stall?

**Problem 5: Custom Allocators [20 points]**

You are implementing an implicit allocator that relies on an eight-byte header, just as it did in **assign6**. The size of a node's payload in bytes is always a multiple of 8, with a minimum of 16, and the payload size—that is, the size of the full node minus the 8 bytes set aside for the header—is stored in the header as a **size_t**. As all payload sizes are multiples of 8, we're free to use the lowest three bits in any way we choose, though we really only use the least significant one to record a node's allocation status. In particular, when the least significant bit within a header equals 0, then the relevant node is allocated. Otherwise, the node is free.




The miniature heap above is all of 120 bytes and divvied up into three nodes, only one of which—the one with a base address 0x107018, is in use. That implies the client is storing that middle node's payload address of 0x107020 in a variable somewhere and using its 48 bytes to store whatever it wants in there.

Assume the following types, constants, and global variables have been defined, and all global variables have been initialized.

```
typedef size_t Header;

#define HEADER_SIZE sizeof(Header)
#define ALIGNMENT sizeof(size_t)
#define MIN_PAYLOAD_SIZE (2 * sizeof(size_t))
#define MIN_NODE_SIZE (HEADER_SIZE + MIN_PAYLOAD_SIZE)

static void *segment_start;
static void *segment_end;

#define IS_FREE_MASK (1UL << 0)

#define ALLOCATED 0
#define FREE 1
```

For this problem, you'll be asked to implement a collection of utility functions that might be used for either a more sophisticated implicit allocator than the one you implemented in **assign6**, or a slightly less elaborate explicit allocator than the one you implemented in **assign6**.

a) [6 points, 2 and 4] Implement two utility functions—**set_payload_size** and **set_free_status**—that know how to update the payload size and allocation status within a node's header. In particular, **set_payload_size** overwrites the upper 63 bits with the upper 63 bits of the supplied size while leaving the allocation bit alone, and **set_free_status** updates the least significant bit while leaving the other 63 bits as is.

```
void set_payload_size(Header *header, size_t size) {
    // assume size >= MIN_PAYLOAD_SIZE and a multiple of ALIGNMENT



}

void set_free_status(Header *header, size_t status) {
    // assume status is either FREE or ALLOCATED









}
```

b) [3 points] Implement a function called **right_neighbor**, which accepts the address of a valid header and returns the address of the **right neighbor's header**, or **NULL** if there is no such neighbor. Referring to the prior diagram, a call to **right_neighbor(0x107018)** would return **0x107050**, and a call to **right_neighbor(0x107050)** would return **NULL**. For simplicity, you may assume the existence of self-explanatory utility functions with the following prototypes:

```
bool is_free(Header *header);
size_t get_payload_size(Header *header);

Header *right_neighbor(Header *header) {









}
```

c) [4 points] Implement a function called **right_coalesce** which accepts the address of a node header and, provided the right neighbor node exists and is free, absorbs the entirety of that right neighbor as additional payload, updating the header of the merged node to retain its own allocation status while updating its payload size as appropriate. Your **right_coalesce** function should return **true** if and only if the right neighbor existed and was free to be coalesced, and **false** otherwise.

```
bool right_coalesce(Header *header) {




}
```

**d)** [7 points] Finally, implement the **implicit_to_explicit** function, which visits every single node in the heap, threads a doubly linked list through the payloads of all free nodes, and returns the address of the leading free payload. Since all payloads are at least 16 bytes in size, the first 16 bytes of each can be recruited to store two, eight-byte **payload** pointers. You should use the supplied **payload** type to layer over the first 16 bytes of each free payload so it can be more easily incorporated into a growing doubly linked list. You may, however, order the payloads within the explicit free list in any way that's convenient.

```
typedef struct payload {
   struct payload *prev;
   struct payload *next;
} payload;

payload *implicit_to_explicit() {
   // start from the segment_start global




}
```