University of Washington – Computer Science & Engineering

CSE 333 A & B Autumn 2025 Midterm: Version A

Last Name:	Lastname
First Name:	Firstname
Student ID Number:	123456789
UWNetID:	netid
All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE333 who haven't taken it yet. Violation of these terms could result in a failing grade. **(please sign)**	*Firstname Lastname*

Do not turn the page until 5:35 pm.

Instructions

- This exam contains 22 pages, including this cover page. Put your final answers in the boxes and blanks provided. You may make use of the 'overflow box' on page 13 for additional answer space.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed one 5"x8" index card (double-sided) of *handwritten* notes.
- Please silence and put away all cell phones and other mobile or noise-making devices.
- You have 50 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read *all* questions first and start where you feel the most confident.
- Relax. You are here to learn.

Question	1	2	3	4	5	6	Total
Possible Points	9	7	12	13	12	1	**54**

1. Half Baked (C Preprocessor)

Suppose you have the following four C source files:

baker.c

```
#include "ingredients.h"
#include "recipe.h"

grams_t measure(int scoops) {
    return SUGAR * scoops;
}
```

ingredients.h

```
#ifndef INGREDIENTS_H_
#define INGREDIENTS_H_

typedef double grams_t;
grams_t measure(int scoops);

#endif
```

recipe.h

```
#ifndef RECIPE_H_
#define RECIPE_H_

#include "ingredients.h"
#define OVEN_TEMP 350
#define SUGAR OVEN_TEMP / 10
#define MIX(x) x + 3

#endif
```

main.c

```
#include "ingredients.h"
#include "recipe.h"

int printf(const char *format, ...);

int main() {
    int scoops = SUGAR;
    grams_t batter = measure(
        scoops + OVEN_TEMP);
    printf("%f\n", batter)
    printf("%d\n", MIX(2 * 3));
    return 0;
}
```

a) Show the resulting code for main.c after it is processed by the C **preprocessor** (*i.e.,* the result of the command `cpp -E -P main.c`).

Hint: The C preprocessor does not analyze the C code it produces for correctness.

```
typedef double grams_t;
grams_t measure(int scoops);
int printf(const char *format, ...);
int main() {
    int scoops = 350 / 10;
    grams_t batter = measure(
              scoops + 350);
    printf("%f\n", batter)
    printf("%d\n", 2 * 3 + 3);
    return 0;
}
```

b) You attempt to create an executable using the following command:

```
gcc -Wall -g -std=c17 -o main main.c
```

If the executable *main* is successfully created, what is its output to the console when run (*i.e.,* what is 'printed' to standard output)? If an executable is not successfully created then explain the problem(s) briefly and the changes necessary to produce an executable successfully.

Problem 1: missing semicolon after first invocation of 'printf' in main.c

Solution 1: add missing semicolon

Problem 2: the measure function is declared, but not defined

Solution 2: add baker.c to the gcc call so it is compiled and linked into executable, or separately compile baker.c into an object file and link it into final executable for main

2. Pizza My Heart (C++ classes)

Assume the following code compiles and executes with no errors:

pizza.cc

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Pizza {
public:
    Pizza() : slices_(8) { cout << "Bake whole pizza\n"; }
    Pizza(int s) : slices_(s) {
        cout << "Bake " << slices_ << " slice pizza\n";
    }
    Pizza(const Pizza& p) : slices_(p.slices_) {
```

```
        cout << "Clone " << slices_ << " slice pizza\n";
    }
    Pizza& operator=(const Pizza& p) {
        cout << "Replace " << slices_ << " slice pizza with "
             << p.slices_ << " slice pizza\n";
        if (this != &p) slices_ = p.slices_;
        return *this;
    }
    ~Pizza() {
        cout << "Eat " << slices_ << " slice pizza\n";
    }
    int slices() const {
        return slices_;
    }

private:
    int slices_;
};

Pizza deliver(Pizza p) {
    cout << "Delivering pizza...\n";
    return p;
}
int main() {
    Pizza a(8);
    Pizza b(4);
    cout << "--1--\n";
    b = a;

    cout << "--2--\n";
    Pizza c = deliver(a);

    cout << "--3--\n";
    return EXIT_SUCCESS;
}
```

In the space below, write what is printed to standard output. You should assume that all copy constructors, constructors, assignment operators, and destructors are called as specified by the C++ language and not eliminated by possible compiler optimizations.

```
Bake 8 slice pizza
Bake 4 slice pizza
--1--
Replace 4 slice pizza with 8 slice pizza
--2--
Clone 8 slice pizza
Delivering pizza...
Clone 8 slice pizza
Eat 8 slice pizza
--3--
Eat 8 slice pizza
Eat 8 slice pizza
Eat 8 slice pizza
```

```
// C++17 (or later) standard (FULL marks)
// See Ed post for why this is no longer
// considered copy elision or an
// optimization
g++ -fno-elide-constructors -Wall -g
-std=c++17 -O0 -o pizza pizza.cc
```

```
Bake 8 slice pizza
Bake 4 slice pizza
--1--
Replace 4 slice pizza with 8 slice pizza
--2--
Clone 8 slice pizza
Delivering pizza...
Clone 8 slice pizza
Clone 8 slice pizza
Eat 8 slice pizza
Eat 8 slice pizza
--3--
Eat 8 slice pizza
Eat 8 slice pizza
Eat 8 slice pizza
```

```
// C++11 standard (FULL marks)
g++ -fno-elide-constructors -Wall -g
-std=c++11 -O0 -o pizza pizza.cc
```

3. What's The Point? (Memory Diagrams)

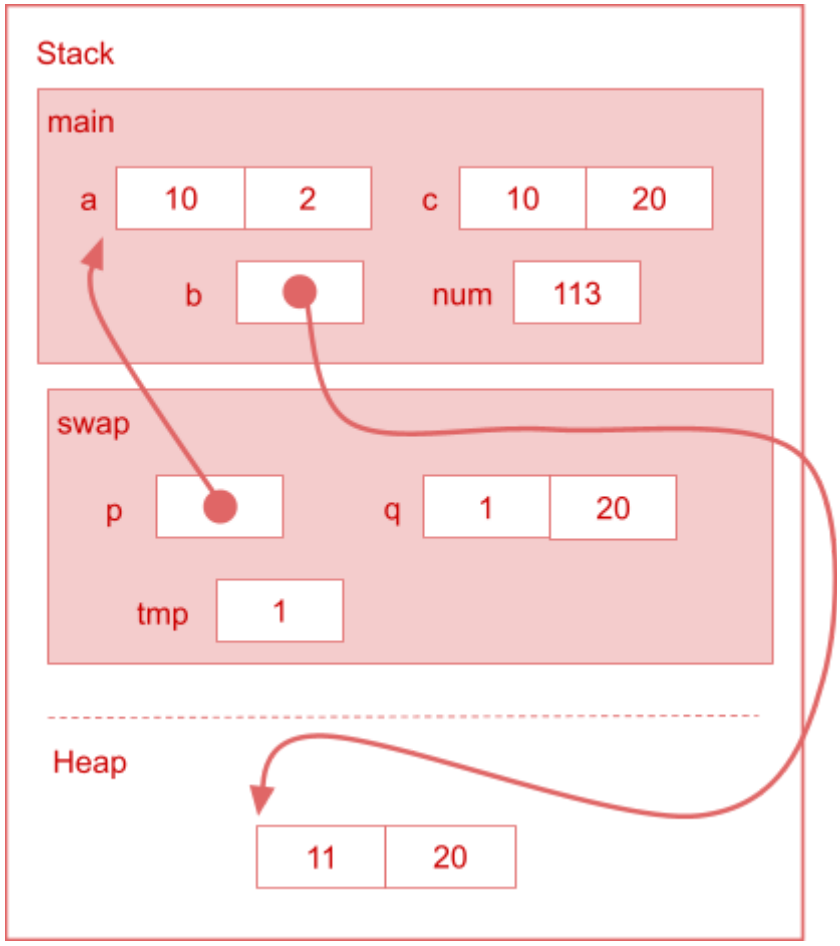
Assume the following code compiles and executes with no errors:

point.c

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

typedef struct point {
    int x, y;
```

```
} Point;

Point* NewPoint(int x, int y) {
    Point* r = (Point*) malloc(sizeof(Point));
    r->x = x;
    r->y = y;
    return r;
}

int Shift(Point p, Point* r) {
    p.y += 100;
    r->x += 1;
    return p.y + r->x;
}

void Swap(Point* p, Point q) {
    int tmp = p->x;
    p->x = q.x;
    q.x = tmp;
    // ===== DRAW MEMORY DIAGRAM HERE =====
    printf("%d %d\n", p->x, q.x);
}

int main(void) {
    Point a = {1, 2};
    Point* b = NewPoint(10, 20);
    Point c = *b;

    int num = Shift(a, b);
    Swap(&a, c);

    printf("%d\n", num);
    printf("%d %d\n", a.x, a.y);
    printf("%d %d\n", b->x, b->y);

    free(b);
    return EXIT_SUCCESS;
}
```

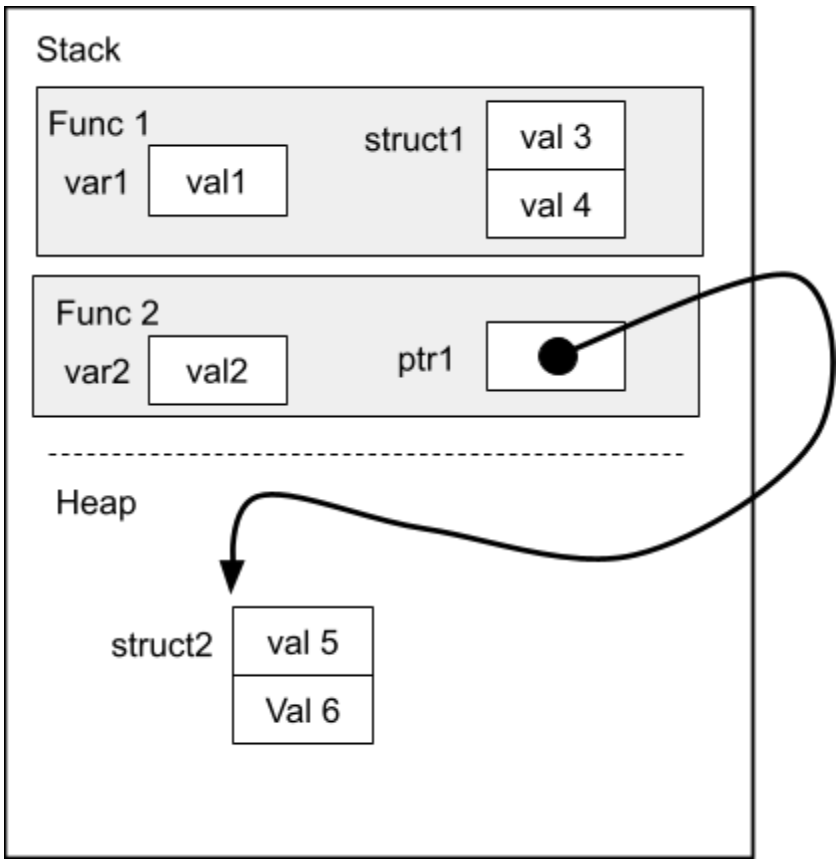
a) Draw a memory diagram of the same format as the example on the following page, showing the content of process memory when execution reaches the comment "`// ===== DRAW MEMORY DIAGRAM HERE =====`". Make sure to carefully distinguish between local stack variables and heap-allocated memory and to have boxes for the stack frames of each relevant function.



b) When executed what does this program write to standard output on the console?

```
10 1
113
10 2
11 20
```

Example memory diagram, of the form you'll draw in problem 3a. **NOTE:** Your solution may have completely different variables, pointers and stack frames than this drawing! It is just showing you *how* to draw such information.



4. Ring Around the Rosie, a Pocket Full of POSIX (POSIX I/O)

a) The following program is designed to read from a source file and write to a destination file, up to 100 bytes at a time, until the entire contents have been copied. For simplicity, recoverable system call errors such as EINTR and EAGAIN are not handled. Fill in the appropriate blanks so that the program executes as intended while avoiding leaking system resources.

Caution: Not all blanks need to be used.

posix.c

```
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h> // For file permissions

#define BUF_SIZE 100

int main(int argc, char** argv) {
    if (argc != 3) {
        fprintf(stderr, "usage: %s SRC_FILE DST_FILE\n", argv[0]);
        return EXIT_FAILURE;
    }
    int input_fd = open(argv[1], O_RDONLY);
    if (input_fd < 0) {
        perror("Couldn't open input file.");
        // first blank unnecessary
        // second blank unnecessary
        return EXIT_FAILURE;
    }
    // Create file if needed, truncate existing, give user RW permissions
    int output_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC,
                                  S_IRUSR | S_IWUSR);
    if (output_fd < 0) {
        perror("Couldn't open output file.");
        close(input_fd);
        // second blank unnecessary
        return EXIT_FAILURE;
    }
    char buffer[BUF_SIZE];
    ssize_t bytes_read;
```

```
    while ((bytes_read = read(input_fd, buffer, BUF_SIZE)) > 0) {
        ssize_t bytes_written = 0;
        while (bytes_written < bytes_read) {
            ssize_t result = write(output_fd, buffer + bytes_written,
                                   bytes_read - bytes_written);
            if (result < 0) {
                perror("Error copying.");
                close(input_fd);
                close(output_fd);
                return EXIT_FAILURE;
            }
            bytes_written += result;
        }
    }
    if (bytes_read < 0) {
        perror("Error processing entire file.");
        close(input_fd);
        close(output_fd);
        return EXIT_FAILURE;
    }
    close(input_fd);
    close(output_fd);
    return EXIT_SUCCESS;
}
```

b) Describe a scenario, if any, when buffered I/O is preferable to unbuffered I/O. Justify your answer.

Scenario: a workload that produces a large number of small outputs to a file.

Justification: buffered output can be substantially more efficient in this scenario as each write to a file will invoke a (relatively slow) system call to complete a write to disk / storage.

c) Describe a scenario, if any, when unbuffered I/O is preferable to buffered I/O. Justify your answer.

Scenario: an application that must record information to a file as soon as possible to avoid data loss.

Justification: if power loss occurs, it's more likely data would be lost when using buffered I/O which will not necessarily flush the buffer to the file until the buffer is filled (or explicitly flushed).

5. The Forever Purge (HW1 extension)

Beginning with the LinkedList (LL) implementation from HW1 we would like to add a new function to its public interface. This new function should remove all elements from the LL that match the given payload and return the number of elements removed. The function should adhere to the following specification:

Function declaration (prototype) to be added into LinkedList.h

```
// Return the number of elements removed.
// ll - LinkedList to search
// payload - Elements containing this payload will be purged from ll
// May assume that the ll is valid, but may have zero elements
// May assume that you are passed a valid pointer to a Payload_Compare()
// function that returns 0 if the elements match
int LinkedList_Purge(LinkedList* ll, LLPayload_t payload,
                     LLPayloadFreeFnPtr payload_free_function,
                     int(*Payload_Compare)(LLPayload_t, LLPayload_t));
```

Give an implementation of LinkedList_Purge as it would appear in LinkedList.c. A copy of the LinkedList.h and LinkedList_priv.h files are included at the end of the exam, and you may remove them from the exam to use for this question. They will not be scanned for grading. Your code may allocate new data structures while it is executing, but should not allocate more data than needed and should not cause any memory leaks. You may assume that any linked list functions that you call will succeed, and you do not need to check for errors when you do that.

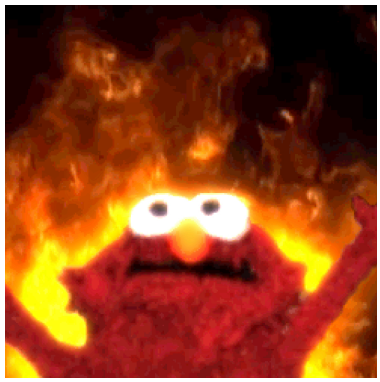
Complete the function definition to be added into LinkedList.c

```
int LinkedList_Purge(LinkedList* ll, LLPayload_t payload,
                     LLPayloadFreeFnPtr payload_free_function,
                     int(*Payload_Compare)(LLPayload_t, LLPayload_t)) {




  // Solution contained entirely on p12 for easier review.
```

```
  Verify333(ll != NULL); // <-- Not required in solution
  LLIterator* lli = LLIterator_Allocate(ll);
  int num_purged = 0;

  while (LLIterator_IsValid(lli)) {
    LLPayload_t curpayload;
    LLIterator_Get(lli, &curpayload);

    if ((*Payload_Compare)(payload, curpayload) == 0) {
      LLIterator_Remove(lli, payload_free_function);
      num_purged += 1;
    } else {
      LLIterator_Next(lli);
    }
  }
  LLIterator_Free(lli);
  return num_purged;
}
```

6. Morale Booster

[1 pt; All non-empty answers receive this point] Select one feature of C or C++. Describe or draw an emoji representing said feature.

Cse333 : 

Extra Work (“Overflow Box”):

You may put additional answers here so long as you indicate which question the work belongs to and indicate in the original answer box that you put an answer in the ‘overflow box 1’.

Reference Material – These Pages NOT Scanned

Useful function reference related to POSIX I/O

```
#include <fcntl.h>

// Opening a file; returns file descriptor, -1 on error
int open(const char *path, int flags, ...
                           /* mode_t mode */ );
```

```
#include <unistd.h>

// Closing a file
int close(int fd);

// Read from a file; returns bytes read, 0 at EOF, -1 on error
ssize_t read(int fd, void buf[count], size_t count);

// Write to a file; returns bytes written, -1 on error
ssize_t write(int fd, const void buf[count], size_t count);
```

```
#include <stdio.h>

// Print a system error message
void perror(const char *s);
```

Reference Material – These Pages NOT Scanned

LinkedList.h

```
#ifndef HW1_LINKEDLIST_H_
#define HW1_LINKEDLIST_H_

#include <stdbool.h>    // for bool type (true, false)
#include <stdint.h>     // for uint64_t, etc.

////////////////////////////////////////////////////////////////////////////
// A LinkedList is a doubly-linked list.
//
// We provide the interface to the LinkedList here; your job is to fill
// in the implementation holes that we left in LinkedList.c.
//
// To hide the implementation of LinkedList, we declare the "struct ll"
// structure and its associated typedef here, but we *define* the structure
// in the internal header LinkedList_priv.h. This lets us define a pointer
// to LinkedList as a new type while leaving the implementation details
// opaque to the customer.
typedef struct ll LinkedList;

// LLPayload type definition:
// For generality, a payload must be large enough to hold a pointer.
// If the client's data is no bigger than a pointer, a copy of that
// data can be stored in the LinkedList, by casting it to the LLPayload
// type.  Otherwise, a pointer to the client's data is maintained in
// the list.
typedef void* LLPayload_t;

// When a customer frees a linked list or a contained node, they need
// to pass in a pointer to a function which does any necessary freeing
// of the payload. We invoke the pointed-to function once for each node
// requiring freeing.
//
// Additional Note: This is a function pointer. Please refer to the end of
// Lecture 3 slides (Pointers, pointers, pointers...) for more detail on the
// syntax and usage.
typedef void(*LLPayloadFreeFnPtr)(LLPayload_t payload);
```

```
// Allocate and return a new linked list.  The caller takes responsibility for
// eventually calling LinkedList_Free to free memory associated with the list.
//
// Arguments: none.
// Returns:
// - the newly-allocated linked list (never NULL).
LinkedList* LinkedList_Allocate(void);

// Free a linked list that was previously allocated by LinkedList_Allocate.
//
// Arguments:
// - list: the linked list to free.  It is unsafe to use "list" after this
//   function returns.
// - payload_free_function: a pointer to a payload freeing function; see above
//   for details on what this is.
void LinkedList_Free(LinkedList *list,
                     LLPayloadFreeFnPtr payload_free_function);

// Return the number of elements in the linked list.
//
// Arguments:
// - list:  the list to query.
// Returns:
// - list length.
int LinkedList_NumElements(LinkedList *list);

// Adds a new element to the head of the linked list.
//
// Arguments:
// - list: the LinkedList to push onto.
// - payload: the payload to push; it's up to the caller to interpret and
//   manage the memory of the payload.
void LinkedList_Push(LinkedList *list, LLPayload_t payload);

// Pop an element from the head of the linked list.
//
// Arguments:
// - list: the LinkedList to pop from.
// - payload_ptr: a return parameter; on success, the popped node's payload
//   is returned through this parameter.
```

```
// Returns:
// - false on failure (eg, the list is empty).
// - true on success.
bool LinkedList_Pop(LinkedList *list, LLPayload_t *payload_ptr);

// Adds a new element to the tail of the linked list.
//
// This is the "tail" version of LinkedList_Push.
//
// Arguments:
// - list: the LinkedList to push onto.
// - payload: the payload to push; it's up to the caller to interpret and
//   manage the memory of the payload.
void LinkedList_Append(LinkedList *list, LLPayload_t payload);

// When sorting a linked list or comparing two elements of a linked list,
// customers must pass in a comparator function.  The function accepts two
// payloads as arguments and returns an integer that is:
//    -1  if payload_a < payload_b
//     0  if payload_a == payload_b
//    +1  if payload_a > payload_b
//
// Additional Note: This is a function pointer. Please refer to the end of
// Lecture 3 slides (Pointers, pointers, pointers...) for more detail on the
// syntax and usage.
typedef int(*LLPayloadComparatorFnPtr)(LLPayload_t payload_a,
                                       LLPayload_t payload_b);

// Sorts a LinkedList in place.
//
// Arguments:
// - list: the list to sort.
// - ascending: if false, sorts descending; else sorts ascending.
// - comparator_function:  this argument is a pointer to a payload comparator
//   function; see above.
void LinkedList_Sort(LinkedList *list, bool ascending,
                     LLPayloadComparatorFnPtr comparator_function);

////////////////////////////////////////////////////////////////////////////
// Linked list iterator.
//
```

```
// Linked lists support the notion of an iterator, similar to Java iterators.
// You use an iterator to navigate forward through the linked list and remove
// elements from the list.  You use LLIterator_Allocate() to manufacture a new
// iterator and LLIterator_Free() to free an iterator when you're done with it.
//
// If you use a LinkedList*() function to mutate a linked list, any iterators
// you have on that list become undefined (ie, dangerous to use; arbitrary
// memory corruption can occur). Thus, you should only use LLIterator*()
// functions in between the manufacturing and freeing of an iterator.
typedef struct ll_iter LLIterator;  // same trick to hide implementation.

// Manufacture an iterator for the list.  Caller is responsible for
// eventually calling LLIterator_Free to free memory associated with
// the iterator.
//
// Arguments:
// - list: the list from which we'll return an iterator.
//
// Returns:
// - a newly-allocated iterator, which may be invalid or "past the end" if
//   the list cannot be iterated through (eg, empty).
LLIterator* LLIterator_Allocate(LinkedList *list);

// When you're done with an iterator, you must free it by calling this
// function.
//
// Arguments:
// - iter: the iterator to free. Don't use it after freeing it.
void LLIterator_Free(LLIterator *iter);

// Tests to see whether the iterator is pointing at a valid element.
//
// Arguments:
// - iter: the iterator to test.
//
// Returns:
// - true: if iter is not past the end of the list.
// - false: if iter is past the end of the list.
bool LLIterator_IsValid(LLIterator *iter);

// Advance the iterator, i.e. move to the next node in the list.  The
```

```
// passed-in iterator must be valid (eg, not "past the end").
//
// Arguments:
// - iter: the iterator.
// Returns:
// - true: if the iterator has been advanced to the next node.
// - false: if the iterator is no longer valid after the
//   advancing has completed (eg, it's now "past the end").
bool LLIterator_Next(LLIterator *iter);

// Returns the payload of the list node that the iterator currently points
// at.  The passed-in iterator must be valid (eg, not "past the end").
//
// Arguments:
// - iter: the iterator to fetch the payload from.
// - payload: a "return parameter" through which the payload is returned.
void LLIterator_Get(LLIterator *iter, LLPayload_t *payload);

// Remove the node the iterator is pointing to.  After deletion, the iterator
// may be in one of the following three states:
// - if there was only one element in the list, the iterator is now invalid
//   and cannot be used.  In this case, the caller is recommended to free
//   the now-invalid iterator.
// - if the deleted node had a successor (eg, the deleted node was the head),
//   the iterator is now pointing at its successor.
// - if the deleted node was the tail, the iterator is now pointing at the
//    predecessor.
//
// The passed-in iterator must be valid (eg, not "past the end").
//
// Arguments:
// - iter:  the iterator to delete from.
// - payload_free_function: invoked to free the payload.
//
// Returns:
// - false if the deletion succeeded, but the list is now empty.
// - true if the deletion succeeded, and the list is still non-empty.
bool LLIterator_Remove(LLIterator *iter,
                       LLPayloadFreeFnPtr payload_free_function);

#endif  // HW1_LINKEDLIST_H_
```

LinkedList_priv.h

```
#ifndef HW1_LINKEDLIST_PRIV_H_
#define HW1_LINKEDLIST_PRIV_H_

#include "./LinkedList.h"  // for LinkedList and LLIterator

// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// Internal structures and helper functions for our LinkedList implementation.
//
// These would typically be located in LinkedList.c; however, we have broken
// them out into a "private .h" so that our unittests can access them.  This
// allows our test code to peek inside the implementation to verify correctness.
//
// Customers should not include this file or assume anything based on
// its contents.
// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


// A single node within a linked list.
//
// A node contains next and prev pointers as well as a customer-supplied
// payload pointer.
typedef struct ll_node {
  LLPayload_t     payload;  // customer-supplied payload pointer
  struct ll_node *next;     // next node in list, or NULL
  struct ll_node *prev;     // prev node in list, or NULL
} LinkedListNode;

// The entire linked list.
//
// We provided a struct declaration (but not definition) in LinkedList.h;
// this is the associated definition.  This struct contains metadata
// about the linked list.
typedef struct ll {
  int               num_elements;  //  # elements in the list
  LinkedListNode   *head;  // head of linked list, or NULL if empty
  LinkedListNode   *tail;  // tail of linked list, or NULL if empty
} LinkedList;

// A linked list iterator.
//
// We expose the struct declaration in LinkedList.h, but not the definition,
// similar to what we did above for the linked list itself.
typedef struct ll_iter {
  LinkedList       *list;  // the list we're for
  LinkedListNode   *node;  // the node we are at, or NULL if broken
```

```
} LLIterator;


// Remove an element from the tail of the linked list.
//
// This is the "tail" version of LinkedList_Pop, and the converse of
// LinkedList_Append.
//
// Arguments:
// - list: the LinkedList to remove from
// - payload_ptr: a return parameter; on success, the sliced node's payload
//   is returned through this parameter.
// Returns:
// - false: on failure (eg, the list is empty).
// - true: on success.
bool LLSlice(LinkedList *list, LLPayload_t *payload_ptr);

// Rewind an iterator to the front of its list.
//
// Arguments:
// - iter: the iterator to rewind.
void LLIteratorRewind(LLIterator *iter);


#endif  // HW1_LINKEDLIST_PRIV_H_
```

Reference Material – These Pages NOT Scanned