Question 1. (12 points) Making things. Suppose we have the following `Makefile` for a C++ program with the these source files: `A.h`, `A.cc`, `B.h`, `B.cc`, `X.h`, and `main.cc`.

```
all: main

A.o: A.cc A.h X.h
   g++ -Wall -g -c A.cc

B.o: B.cc B.h X.h
   g++ -Wall -g -c B.cc

main.o: main.cc A.h B.h X.h
   g++ -Wall -g -c main.cc

main: A.o B.o main.o
   g++ -Wall -g -o main A.o B.o main.o

clean:
   rm -rf *.o main *~
```

(`-std=c++11` omitted to save space. Recall that the `-c` option causes the compiler to generate a `.o` file only, and the default `.o` file name is taken from the `.cc` file name.)

Suppose we have just executed the command `make` with no arguments.

(a) (4 points) Write the sequence of commands that will be executed if we delete `main.o` and type `make` again. List the commands in the exact order that they will get executed.

```
g++ -Wall -g -c main.cc
g++ -Wall -g -o main A.o B.o main.o
```

(continued on next page)

Question 1. (cont.)

(b) (4 points) Assuming that all files are up to date (we have just typed `make`), list the sequence of commands that will be executed if we modify `B.h` and type `make` again. List the commands in the exact order that they will get executed.

```
g++ -Wall -g -c B.cc
g++ -Wall -g -c main.cc
g++ -Wall -g -o main A.o B.o main.o
```

(c) (4 points) Assuming that all files are up to date (we have just typed `make`), list the sequence of commands that will be executed if we modify `X.h` and type `make` again. List the commands in the exact order that they will get executed.

```
g++ -Wall -g -c A.cc
g++ -Wall -g -c B.cc
g++ -Wall -g -c main.cc
g++ -Wall -g -o main A.o B.o main.o
```

Question 2. (14 points) We have two files, `hdr.h` and `ppro.c`, containing the following C code:

hdr.h:

```
#ifndef _HDR_H_
#define _HDR_H_

#ifdef BAR
#define FOO 4
#else
#define FOO 5
#define BAR 2
#endif

#endif // _HDR_H_
```

ppro.c:

```
#include <stdio.h>
#include "hdr.h"

#define SQR(x) x * x
#define DBL(x) x + x
#define BAR 3

int main(int argc,
                char ** argv) {
  printf("%d %d\n",FOO,BAR);
  int a = SQR(DBL(FOO));
  int b = DBL(SQR(BAR));
  printf("%d %d\n", a, b);
}
```

(a) (10 points) Show the result produced by the C preprocessor when it processes file `ppro.c` (i.e., if we were compiling this file, what output would the preprocessor send to the C compiler that actually translates the program to machine code?). You should ignore the `#include <stdio.h>` directive since that includes library declarations that we do not have access to. Write the rest of the preprocessor output below.

```
int main(int argc, char ** argv) {
  printf("%d %d\n", 5, 3);
  int a = 5 + 5 * 5 + 5;
  int b = 3 * 3 + 3 * 3;
  printf("%d %d\n", a, b);
}
```

(b) (4 points) What output does this program print when it is executed? (It does compile and execute without errors.)

```
5  3
35  18
```

Question 3. (18 points) C programming warmup. Write a program on the next page that works as follows: The program has a single command-line argument giving the name of a text file. The program should open that file and copy its contents to `stdout`, except that when the input file contains two or more successive lines that are identical, then that line should be copied only once. For example, if the input file contains

```
apple
banana
banana
banana
donut
banana
cherry
```

then the program should copy the following lines to `stdout`:

```
apple
banana
donut
banana
cherry
```

You must read input lines by calling a function to read a complete line each time, not by reading one character at a time. The program should terminate with `EXIT_SUCCESS` if all is well, or print an appropriate (brief) message and terminate with `EXIT_FAILURE` if problems occur. You may assume:

- The file contains only ASCII text and each line has a `\n` at the end
- No line contains more than 1200 characters, including any `\n` or `\0` at the end
- It's fine to use standard C library routines for file I/O and string handling
- Any necessary headers have already been `#include`d for you
- You can put all the code in a single `main` function if that makes sense (and it probably does)

Hints: `fgets`, `printf`, `strncpy`, etc.

A collection of reference information that might be useful for this question is included on the last two pages of this exam, including summaries of various standard I/O and string handling functions.

Please write your answer on the next page and **remove this page from the exam. This page will not be scanned for grading.**

(continued on next page)

Question 3 (cont.) Write your solution to the problem below. `#includes` and the header for `main` are written for you to save a bit of time.

```
#include <stdio.h>    // printf, scanf, fopen, etc.
#include <string.h>   // string library
#include <stdlib.h>   // EXIT_SUCCESS, EXIT_FAILURE
// add any additional preprocessor commands you need below
#define BUFSIZE 1200   // size of input buffer
int main(int argc, char ** argv) {
  FILE * text;          // input file
  char line[BUFSIZE];   // current line from input
  char prev[BUFSIZE];   // last line printed to stdout

  // verify that there is one command-line argument
  if (argc != 2) {
    fprintf(stderr, "missing file name\n");
    return EXIT_FAILURE;
  }
  // open file and quit if failure
  text = fopen(argv[1], "r");
  if (text == NULL) {
    fprintf(stderr, "unable to open file\n");
    return EXIT_FAILURE;
  }

  // prev line is initially empty string
  prev[0] = '\0';

  // Read lines - if new one differs, print it and save
  while (fgets(line, BUFSIZE, text) != NULL) {
    if (strncmp(line, prev, BUFSIZE)!= 0) {
      printf("%s", line);
      strncpy(prev, line, BUFSIZE);
    }
  }

  // print message if input terminated because of error
  if (ferror(text)) {
    fprintf(stderr, "error on input");
    return EXIT_FAILURE;
  }
  fclose(text);
  return EXIT_SUCCESS;
}
```

Question 4. (20 points) More deduplication. This question concerns the linked list data structures from the HW1 project. Copies of the header files for `LinkedList` from the project code are provided on separate pages.

We have a client program that is using our `LinkedList` package to store a list of integer values. The integers stored in the list are allocated individually on the heap (each pointer references a separate heap-allocated `int`), and a pointer to each integer is stored in the linked list as the payload pointer in each node. In other words, to append the integer 17 to a list, the client code is doing something like this:

```
int *ptr = (int *)malloc(sizeof(int));
*ptr = 17;
AppendLinkedList(lst, (LLPayload_t)ptr);
```

For this question, write a function `RemoveDuplicates` that deletes adjacent duplicate values from the list, leaving only one copy of each value from each run of duplicate values. So, for example, if the original list contains (pointers to) the following sequence of integers

```
3  5  5  17  9  9  9  5  42  333
```

then `RemoveDuplicates` should leave (pointers to) the following integers on the list:

```
3  5  17  9  5  42  333
```

Any heap data removed from the list should be correctly freed (i.e., no memory leaks).

The full specification of this function is

```
// Replace all runs of duplicate values in a LinkedList
// with a single copy of that value.  Any heap storage
// occupied by the duplicate values is freed.
//
// Arguments:
//
// - ll: the list to examine
//
// Return 0 on success, otherwise return -1 if any error
// int RemoveDuplicates(LinkedList ll);
```

If necessary, you should write appropriate auxiliary functions if needed to free any necessary heap data or for other purposes. For full credit, your solution must use functions in the `LinkedList` package when appropriate and not duplicate code that is already implemented elsewhere. (Specifically, you should use an iterator and related functions to process the list.) Remember that this is a client function, not a part of the `LinkedList` package, and only has access to declarations in `LinkedList.h`.

Please write your answer on the next page and **remove this page from the exam. This page will not be scanned for grading.**

Question 4. (cont.) Complete the implementation of `RemoveDuplicates` below.

```
#include "LinkedList.h"
// other headers omitted to save space
int RemoveDuplicates(LinkedList ll) {
  HWSize_t size = NumElementsInLinkedList(ll);
  if (size > 1) {
    LLIter iter = LLMakeIterator(ll, 0);
    if (iter == NULL)
      return -1;
    LLPayload_t prev, curr;
    LLIteratorGetPayload(iter, &prev);
    LLIteratorNext(iter);
    for (HWSize_t i = 0; i < size - 1; i++) {
      LLIteratorGetPayload(iter, &curr);
      if (*((int *) prev) == *((int *) curr)) {
        LLIteratorDelete(iter, FreePayload);
      } else {
        LLIteratorNext(iter);
        prev = curr;
      }
    }
    LLIteratorFree(iter);
  }
  return 0;
}

// payload free function
void FreePayload(LLPayload_t payload) {
  free(payload);
}
```

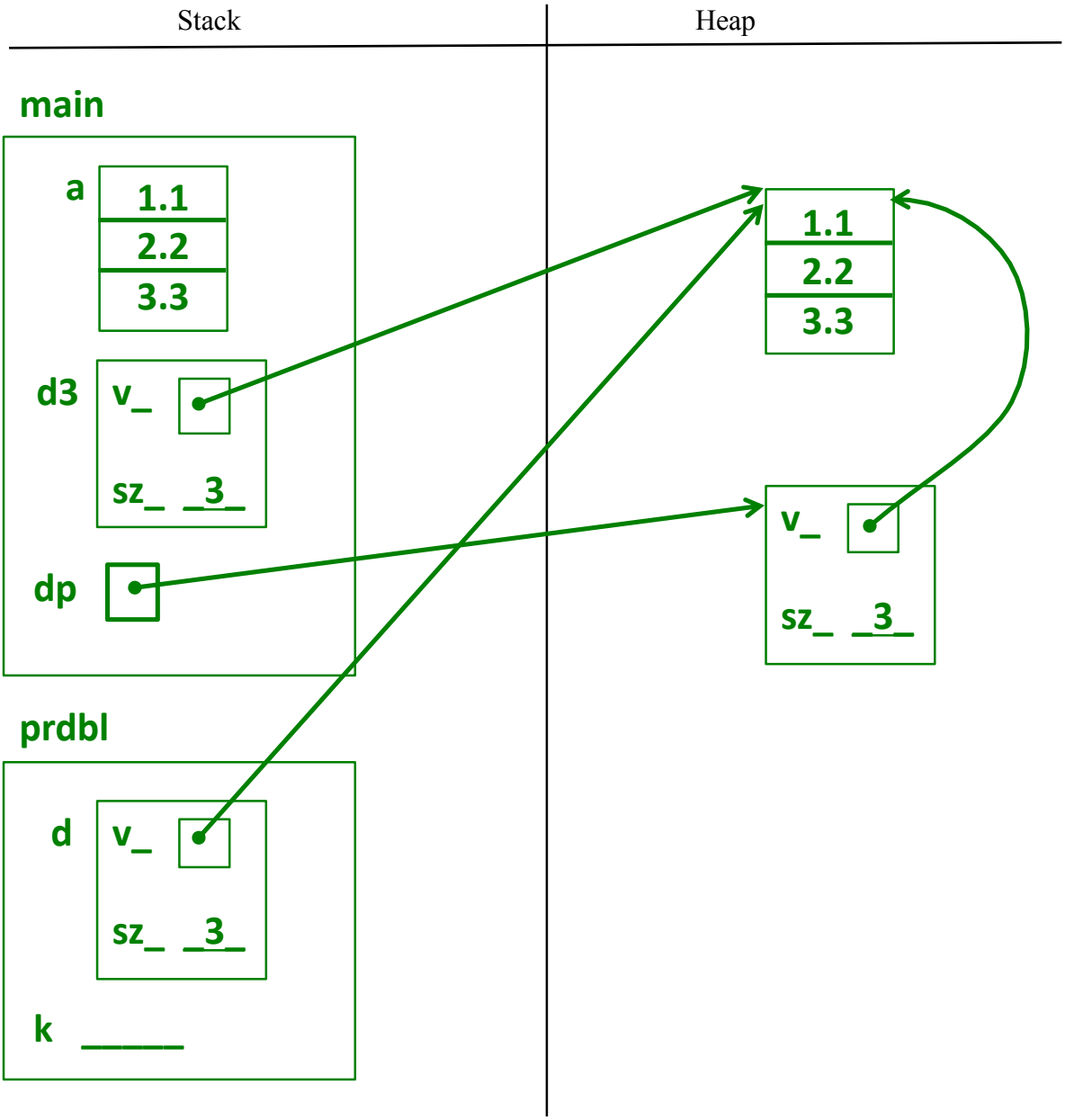
Note: for this particular problem, it would also be possible to use `free` directly as the second argument to `LLIteratorDelete` rather than defining a separate `FreePayload` function(!).

Question 5. (26 points) One of the summer interns is trying to learn C++ and has written the following class that stores an array of doubles and a main program that uses it.

```
class Doubles {
public:
  // construct Doubles given array and # elements
  Doubles(double *vals, uint32_t size)
         : v_(new double[size]), sz_(size) {
    for (uint32_t k = 0; k < size; k++)
      v_[k] = vals[k];
  }

  // destructor, other standard operations
  ~Doubles() { delete[] v_; }
  Doubles(const Doubles &other) = default;
  Doubles &operator=(const Doubles &other) = default;

  // "getter" functions
  double   get(uint32_t k) const { return v_[k]; }
  uint32_t length() const        { return sz_; }

private:
  double* v_;     // heap-allocated array
  uint32_t sz_;   // size of array
};

// print data in a Doubles object
void prdbl(Doubles d) {
  ///// ***>>>> here <<<<*** /////
  cout << "[ ";
  for (uint32_t k = 0; k < d.length(); k++)
    cout << d.get(k) << " ";
  cout << "]" << endl;
}

int main() {
  double a[] = { 1.1, 2.2, 3.3 };
  Doubles d3(a,3);
  Doubles* dp = new Doubles(d3);
  prdbl(d3);
  prdbl(*dp);
  delete dp;
  return 0;
}
```

Please answer the questions about this class on the next page and **remove this page from the exam. This page will not be scanned for grading.**

Question 5. (cont.) (a) (12 points) Draw a *precise* diagram showing the contents of memory the **first time** execution reaches the comment ///// ***>>>> here <<<<*** ///// at the beginning of function `prdbl`. Your diagram should clearly show the contents of the individual stack frames for `main` and `prdbl` and the contents of heap storage, with appropriate arrows from pointers to values that they reference. Then continue with the question on the next page.



(continued on next page)

Question 5. (cont.) (b) (3 points) When the program is executed it crashes. Exactly where does it crash, when, and why? (what is the problem?) (Be brief but precise!)

The code crashes on exit from the *second* call to `prdbl` when it attempts to delete the array of `doubles` on the heap a second time.

There are multiple `Doubles` objects, all of which share the same array of `doubles` on the heap because the default copy constructor does a shallow copy of the object data and does not create a new array for each object. This includes the call-by-value objects created when `prdbl` is called.

When these objects are deleted the destructor deletes the array on the heap resulting in dangling pointers for all other objects constructed from the original one (`d3`). The second time a `Doubles` object is deleted, a double delete error occurs, and this happens when the local parameter object is deleted at the end of the second call to `prdbl`.

(Grading note: explanations did not need to be this detailed for full credit as long as they pinpointed the exact problem and location.)

(c) (3 points) Our summer intern has been googling and thinks that something called the "Rule of 3" is the reason for the crash. The intern proposes replacing the destructor with the following code to match the copy constructor and assignment:

```
~Doubles() = default;
```

Will the program run without crashing if this is done? Why or why not? (briefly)

Yes, it will run without crashing since the array that is shared will never be deleted. There will be a memory leak, because the heap array is never deleted, but there won't be double-delete errors.

Question 5. (cont.) (d) (8 points) What really needs to be done to fix this class so it works properly and behaves appropriately for a C++ class? Give the changes needed below by listing which functions (methods) need to be changed in the original code and writing the correct code below.

We should create a proper copy constructor and assignment operator for `Doubles` so that each instance of the class has its own private copy of the array. Replace the `default` versions with the following or something equivalent:

```
Doubles(const Doubles &other)
  : v_(new double[other.sz_]), sz_(other.sz_) {
  for (uint32_t k = 0; k < sz_; k++)
    v_[k] = other.v_[k];
}

Doubles &operator=(const Doubles &other) {
  if (this == &other)
    return *this;
  delete [] v_;
  v_ = new double[other.sz_];
  sz_ = other.sz_;
  for (uint32_t k = 0; k < sz_; k++)
    v_[k] = other.v_[k];
  return *this;
}
```

Question 6. (8 points) C++ references and assignment. We've claimed that the assignment operator for a C++ class needs to return a reference so that assignment chaining works properly. In other words, the assignment operator for class `Thing` should have the following declaration

```
Thing &operator=(const Thing &other);
```

But our summer intern has discovered that assignment seems to "work" if the first reference is omitted and the declaration looks like this:

```
Thing operator=(const Thing &other);
```

(a) (4 points) Suppose we have three `Things` `t1`, `t2` and `t3`. Why is it that the assignment statement `t1 = t2 = t3;` appears to "work" if the assignment operator result type is `Thing` instead of `Thing&`? What exactly happens when this assignment is executed?

Without the `&` in the return type, we will return a temporary copy of the object instead of a reference to the object that was just assigned. That will appear to "work" because the temporary `Thing` returned by `t2=t3`, which is a copy of `t2`, will be used as the source value in the assignment to t1. (This is true, however, only as long as the temporary copy is constructed in a way that doesn't cause other errors or have visible side effects.)

(b) (4 points) Tricky. Assignment chaining is supposed to allow the assignment

```
(t1 = t2) = t3;
```

to work properly. Does it work correctly if the assignment operator result type is `Thing` instead of `Thing&`? Why or why not? (Hint: consider what assignment should do if we used integers: `int x = 1; int y = 2; int z = 3; (x = y) = z;`. What should that do, and does it work properly with objects if we don't use the correct result type for `operator=`?)

This will not work. The effect of `(x=y)=z;` is to assign `y` to `x`, then assign z to the result of that assignment, which should be a reference to `x`.

If `(x=y)` returns an anonymous temporary, which it does without the `&` reference return type, then the value of `z` will be assigned to that temporary, not to `x`, which will still retain the value assigned to it by `x=y`.

Question 7. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

The traditional last midterm question.

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

All answers received full credit

(b) (1 point) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

None of the above. My answer is _____.

Reference information. Here is a collection of information that might, or might not, be useful while taking the test. You can remove this page from the exam if you wish.

Please do not write on this page. It will not be scanned for grading.

Memory management (<stdlib.h>)

- void * malloc(size_t size)
- void free(void *ptr)
- void * calloc(size_t number, size_t size)
- void * realloc(void *ptr, size_t size)

Strings and characters (<string.h>, <ctype.h>)

Some of the string library functions:

- char* strncpy(*dest*, *src*, *n*), copies exactly *n* characters from *src* to *dst*, adding '\0's at end if the '\0' at the end of the string *src* is found before *n* chars copied.
- char* strcpy(*dest*, *src*)
- char* strncat(*dest*, *src, n*), Appends the first *n* characters of *src* to *dst*, plus a terminating null-character. If the length of the C string in *src* is less than *n*, only the content up to the terminating null-character is copied.
- char* strcat(*dest*, *src*)
- int strncmp(*string1*, *string2*, *n*), <0, =0, >0 if compare <, =, >
- int strcmp(*string1*, *string2*)
- char* strstr(*string, search_string*)
- int strnlen(*s, max_length*), # characters in *s* not including terminating '\0'
- int strlen(*s*)
- Character tests: isupper(*c*), islower(*c*), isalpha(*c*), isdigit(*c*), isspace(*c*)
- Character conversions: toupper(*c*), tolower(*c*)

Files (<stdio.h>)

Some file functions and information:

- Default streams: stdin, stdout, and stderr.
- FILE* fopen(*filename*, *mode*), modes include "r" and "w"
- char* fgets(*line, max_length, file*), returns NULL if eof or error, otherwise reads up to max-1 characters into buffer, including any \n, and adds a \0 at the end
- size_t fread(buf, 1, count, FILE* f)
- size_t fwrite(buf, 1, count, FILE* f)
- int fprintf(format_string, data…, FILE *f)
- int feof(*file*), returns non-zero if end of *file* has been reached
- int ferror(FILE* f), returns non-zero if the error indicator associated with f is set
- int fputs(*line, file*)
- int fclose(*file*)

A few printf format codes: %d (integer), %c (char), %s (char*)

More reference information, C++ this time. You can also remove this page from the exam. **Please do not write on this page.** It will not be scanned for grading.

C++ strings

If `s` is a string, `s.length()` and `s.size()` return the number of characters in it. Subscripts (`s[i]`) can be used to access individual characters.

C++ STL

- If `lst` is a STL `vector`, then `lst.begin()` and `lst.end()` return iterator values of type `vector<...>::iterator`. STL `lists` and `sets` are similar.
- A STL `map` is a collection of `Pair` objects. If p is a `Pair`, then `p.first` and `p.second` denote its two components. If the `Pair` is stored in a `map`, then `p.first` is the key and `p.second` is the associated value.
- If `m` is a `map`, `m.begin()` and `m.end()` return iterator values. For a `map`, these iterators refer to the `Pair` objects in the `map`.
- If `it` is an iterator, then `*it` can be used to reference the item it currently points to, and `++it` will advance `it` to the next item, if any.
- Some useful operations on STL containers (lists, maps, sets, etc.):
  - `c.clear()` – remove all elements from c
  - `c.size()` – return number of elements in c
  - `c.empty()` – true if number of elements in c is 0, otherwise false
- Additional operations on `vectors`:
  - `c.push_back(x)` – copy x to end of c
- Some additional operations on `maps`:
  - `m.insert(x)` – add copy of `x` to `m` (a key-value pair for a `map`)
  - `m.count(x)` – number of elements with key `x` in `m` (0 or 1)
  - `m[k]` can be used to access the value associated with key `k`. If `m[k]` is read and has never been accessed before, then a <key,value> `Pair` is added to the map with `k` as the key and with a value created by the default constructor for the value type (0 or `nullptr` for primitive types).
- Some additional operations on `sets`
  - `s.insert(x)` – add `x` to `s` if not already present
  - `s.count(x)` – number of copies of `x` in `s` (0 or 1)
- You may use the C++11 `auto` keyword, C++11-style `for`-loops for iterating through containers, and any other features of standard C++11, but you are not required to do so.