CSE 333 Midterm Exam Nov. 3, 2017

Name ____ UW ID# ____________

There are 7 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, closed notes, closed electronics, closed telepathy, open mind.

If you don't remember the exact syntax for something, make the best attempt you can. We will make allowances when grading.

Don't be alarmed if there seems to be more space than is needed for your answers – we tried to include more than enough blank space.

Relax, you are here to learn.

Please wait to turn the page until everyone is told to begin.

Score _________________ / 100

1. ______ / 8	5. ______ / 18
2. ______ / 14	6. ______ / 22
3. ______ / 16	7. _____ / 2
4. ______ / 20	

Note: Please write your answers only on the specified pages. Reference pages and pages with only questions and explanations will not be scanned for grading, and you should feel free to remove them from the exam.

There are two pages of reference information at the end of the exam that may be useful for some of the questions.

There is an extra blank page after the last question if you need additional space for one or more answers. That page will be graded if it is not blank.

Question 1. (8 points) Making things. Consider this (buggy) `Makefile` (the options `-Wall`, `-g`, and `-std=c11` were omitted to save space and do not affect the answers):

```
all: prog
foo.c: foo.c foo.h bar.h
     gcc -c -o foo.o foo.c
bar.c: bar.c foo.h bar.h
     gcc -c -o bar.o bar.c
prog: foo.o bar.o
     gcc -o prog foo.o bar.o
clean:
     rm prog *.o
```

For the following questions, be clear about which files already exist or not, precisely why the particular situation occurs, etc., and assume that `make` does *not* have any special knowledge about compiling C beyond the rules given in this `Makefile`. There are multiple correct answers to some of the questions; you only need to give a single correct one. Be brief and to the point.

(a) (3 points) Describe a situation where this `Makefile` would lead `make` not to recompile or relink something that needs recompiling or relinking.

(b) (3 points) Describe a situation where this `Makefile` would lead `make` to report that it does not know about some target.

(c) (2 points) What is wrong with this `Makefile`? How should it be fixed?

Question 2. (14 points) We have a file `ppro.c` containing the following C code:

```
#include <stdio.h>
#define INCR 2
#define PRINT(x) printf("%d\n", x )
#define COMPUTE(x)   (x * x + INCR)
int compute(int x ) { return ( x * x + INCR ); }
int main() {
  int a = 2;
  int b = 3;
  PRINT( compute(a) );
  PRINT( COMPUTE(a) );
  PRINT( compute(a+b) );
  PRINT( COMPUTE(a+b) );
  return 0;
}
```

(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.

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

Question 3. (16 points) A simple C programming warmup. Give an implementation of the following function, which has an array as a parameter and allocates and returns through an output parameter a new array of the same length. In the new array, element *k* should be the sum of elements 0 through *k* of the input array. Example: if the original array contains { 1, 3, 4, 5 }, the new output array should contain { 1, 4, 8, 13 }.

```
#include <stdlib.h>

// Allocate and return a new array where the kth
// element of the new array is the sum of elements
// 0 through k of the original array
// Arguments:
// - arr: the original array
// - arrlen: number of elements in arr
// - outputarr: return parameter - the address of
//   the newly allocated array is stored here.
// Returns true (1) on success, otherwise false
int RunningSum(int *arr, size_t arraylen, int **outputarr) {











}
```

Question 4. (20 points) Making hash out of pointers. This question concerns data structures from the HW1 project. Copies of the header files for `LinkedLists` and `HashTables` from the project code are provided on separate pages.

Consider the following (possibly buggy) `test` function and the beginning of the `InsertHashTable` function from HW1.

```
int InsertHashTable(HashTable table,
                    HTKeyValue newkeyvalue,
                    HTKeyValue *oldkeyvalue) {
     // Draw diagram when execution reaches HERE
     // Remainder of function omitted...
     ...
}

int test() {
  HashTable ht = AllocateHashTable(2);
  HTKeyValue kv, oldkv;
  kv.key = 1;
  kv.value = (void*)333;
  int result = InsertHashTable(ht, kv, &oldkv);
  return result;
}
```

On the next page draw a *precise* diagram of memory at the point when execution reaches the first statement in the body of function `InsertHashTable` (where the "Draw diagram" comment is in the above code).

The data structure definitions involved are given in the `HashTable` and `LinkedList` header files. Also recall the following:

- `AllocateHashTable(`*n*`)` allocates on the heap a new hash table `ht` with an array of *n* buckets. Each bucket element is initialized with the following statement:
  `ht->buckets[i] = AllocateLinkedList();`
  The initialized hash table `ht` is returned as the result of `AllocateHashTable`.
- `AllocateLinkedList()` creates a new `LinkedList` object `ll` on the heap and sets `ll->num_elements = 0` and `ll->head = ll->tail = NULL` before returning `ll` to the caller as the result of `AllocateLinkedList`. No list nodes are allocated when the list is created.

Draw your diagram on the next page. You may remove this page if you wish. **Please do not write on this page – anything written here will be ignored when grading.**

Question 4. (cont.) Draw your diagram below. Local variables and data on the stack should be drawn on the left, and there should be boxes indicating which memory belongs to function `test` and which belongs to function `InsertHashTable`. Draw data allocated on the heap to the right. Use arrows to show how objects are linked by pointers.

Stack

Heap

Question 5. (18 points) Consider the following C++ class, which implements a simple wrapper around an integer variable, and a main function that uses it. This does compile and execute without errors.

```
#include <iostream>
using namespace std;

class Int {
public:
  Int(): n_(0)           { cout << "default ctr" << endl; }
  Int(int n): n_(n)      { cout << "ctr " << n_ << endl; }
  Int(const Int &other): n_(other.n_)
                         { cout << "copy ctr " << n_ << endl; }
  Int &operator=(const Int & other) {
    cout << "op= " << n_ << "=" << other.n_ << endl;
    if (this == &other) return *this;
    n_ = other.n_;
    return *this;
  }
  ~Int() { cout << "dtr " << n_ << endl; }
private:
  int n_;
};

int main() {
  Int n1 = 1;
  Int n2;
  cout << "---" << endl;
  n2 = n1;
  cout << "---" << endl;
  n1 = 17;
  cout << "---" << endl;
  return EXIT_SUCCESS;
}
```

Answer questions about this code on the next page. You may remove this page if you wish. **Please do not write on this page – anything written here will be ignored when grading.**

Question 5. (cont.) (a) (10 points) What output is printed when this program is executed? (Assume that the compiler generates simple code and does not eliminate any operations or perform similar optimizations on the code.)

(b) (8 points) Now suppose that we delete the `&` symbols from the heading of the assignment operator so that it now looks like this:

```
  Int operator=(const Int other) { ... }
```

No other changes are made to the code. Amazingly enough, the program still compiles and executes with no apparent problems. What output is printed when the program is executed after this change? (As with part (a), assume that the compiler is not clever enough to eliminate any operations or otherwise optimize the code.)

Question 6. (22 points) Recall the small C++ string class from lecture. `Str.h` contains

```
class Str {
 public:
  Str();                // default constructor
  Str(const char *s);  // c-string constructor
  Str(const Str &s);   // copy constructor
  ~Str();               // destructor

  // operations
  int length() const;              // = length of this string
  char * c_str() const;            // = new char* copy of this string
  Str &operator=(const Str &s);   // assignment

  // stream output
  friend std::ostream &operator<<(std::ostream &out, const Str &s);

 private:
  char *st_;  // c-string on heap with data bytes terminated by '\0'
};
```

We would like to create a modified version of `Str`. We've removed the `append` function that was part of the class, and we'd like to add "multiplication" operators. The new `*` operators should work as follows: if `s` is a `Str` object, and `k` is an int, then both `s*k` and `k*s` should create and return a new `Str` object (*not* a pointer to a `Str`) holding `k` copies of `s`. For example, if `s` contains the string "ha", then 3*s should return a `Str` that contains "hahaha". The `*` operator should not modify `s`. In addition there should be a new `*=` assignment operator such that `s*=k` modifies `s` so it now has the value `s*k`. For all of these operators, if `k<=0` then the result should be an empty string with length 0 (i.e., `st_` should point to an array with only a '\0' byte at the end).

(a) (8 points) Write declarations for the new `*` and `*=` operators to be added to the header file `Str.h`. You should show clearly which declarations are added to the class as either member or friend functions, and which declarations are additional overloaded functions that are not part of the class. You should follow best practices in deciding which functions to include in the class and which ones should not be included.

Additional declarations to be added to class `Str` in `Str.h`:

Additional declarations to add to `Str.h` that are not part of class `Str`:

Question 6. (cont) (b) (14 points) Give the code to be added to file `Str.cc` to implement the new `*` and `*=` operators added to `Str.h` on the previous page.

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?

(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 ________________________.

Extra space for answers, if needed. Please be sure to label which question(s) are answered here, and be sure to put a note on the question page so the grader will know to look here.

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 fewer than *n* characters in *src* so that *n* chars. are copied.
- char* strcpy(*dest*, *src*)
- char* strncat(*dest*, *src, n*), append up to *n* characters from *src* to the end of *dest*, put '\0' at end, either copy from *src* or added if no '\0' in copied part of *src*.
- 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 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 `list`s and `set`s 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 `vector`s:
  - `c.push_back(x)` – copy x to end of c
- Some additional operations on `map`s:
  - `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 `set`s
  - `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.