University of Washington – Computer Science & Engineering

CSE 333 Autumn 2019 Midterm Solution: Version A

Last Name: Programmer

First Name: Systems

Student ID Number:

Name of person to your Left | Right

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)** Systems Programmer

Do not turn the page until 11:30.

Instructions

- This exam contains 8 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed one page (US letter, 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.

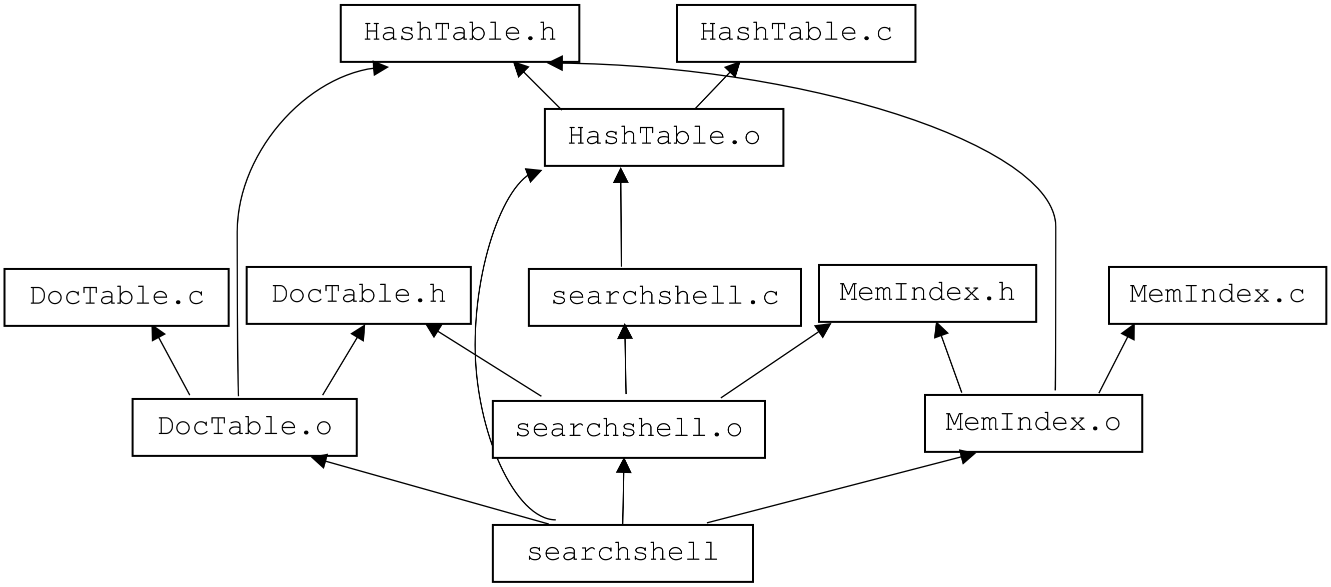
Advice

- 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**	**7**	**8**	**Total**
Possible Points	13	8	12	6	28	19	25	1	**112**

Question 1:

Consider the dependency graph, below, which was derived from our project's Makefile.



(A) [3 pts] If DocTable.h is modified, which targets need to be rebuilt?

DocTable.o, searchshell.o, searchshell

(B) [3 pts] If DocTable.c is modified, which targets need to be rebuilt?

DocTable.o, searchshell

(C) [4 pts] In HW2, MemIndex.c contained a line to #include "DocTable.h". The Makefile snippet which generated our dependency graph is below. What, if anything, needs to change in it?

x Changes Are Required to Makefile (see below) ☐ No Changes Necessary

```
MemIndex.o: MemIndex.c MemIndex.h HashTable.h    DocTable.h
        $(CC) $(CFLAGS) -c $<
```

(D) [3 pts] If changes are necessary to the Makefile, please describe how these changes would impact your answers to (A) and (B).

x Changes Are Required to (A) and (B) (described below) ☐ No Changes Necessary

Part (A) needs to add MemIndex.o

Question 2:

[8 pts] Of the following, which are POSIX system calls and which are not?

	Syscall	Not Syscall
size_t **fwrite**(const void *ptr, size_t size, size_t nmemb, FILE *stream);		x
struct dirent* **readdir**(DIR *dirp);	x	
size_t **strlen**(const char *s);		x
int **close**(int fildes);	x	

Question 3:

[12 pts] Recall that the steps of building and running a program are: preprocessing, compilation, linking, and loading. At which step do each of the following events occur?

Templates are instantiated (eg, `list<double>`) for a specific type	Compilation
Space is reserved for global variables which reside in static data	Linking
Global variables which reside in static data are initialized to their values	Loading
The contents of header files (eg, `stdio.h`) are copied into source (eg, `.c`)	Preprocessing
References to declared-but-not-defined symbols (eg, function declarations and `extern`'ed variables) are resolved	Linking
Source files (eg, `main.cc`) are checked for type errors	Compilation

Question 4:

UW student numbers (**not** UWNetIDs) are 7-digit numbers that uniquely identify every currently- and formerly-enrolled student; the last four digits are a counter. You are designing a file format for storing these IDs on disk, and these files will store at most 200 years' worth of students; there are ~30,000 students per year. What type should you choose to represent these student numbers?

Hint: 2^{16} == 65,536; 2^{32} == 4,294,967,296; 2^{64} == 18,446,744,073,709,551,616

(A) [3 pts] X Signed integer □ Unsigned integer

(B) [3 pts] □ 16-bit integer X 32-bit integer □64-bit integer

Question 5:

This holiday-themed C program has 3 files. Remember that `%` is the modulo or “remainder” operator.

trickortreat.h

```
#ifndef TRICKORTREAT_H_
#define TRICKORTREAT_H_

#define EATEN_CANDY      0
#define CHOCOLATE_BAR    1
#define CANDY_CORN       2
#define LOLLIPOP         3

int Dispense();

#endif  // TRICKORTREAT_H_
```

trickortreat.c

```
#include "trickortreat.h”
#define NUM_CANDY_TYPES  3
#define TO_CANDY(c)      ((c) + 1)

static int kids = 0;

int Dispense() {
  int candy =
    TO_CANDY(kids % NUM_CANDY_TYPES);
  kids++;
  return candy;
}
```

main.c

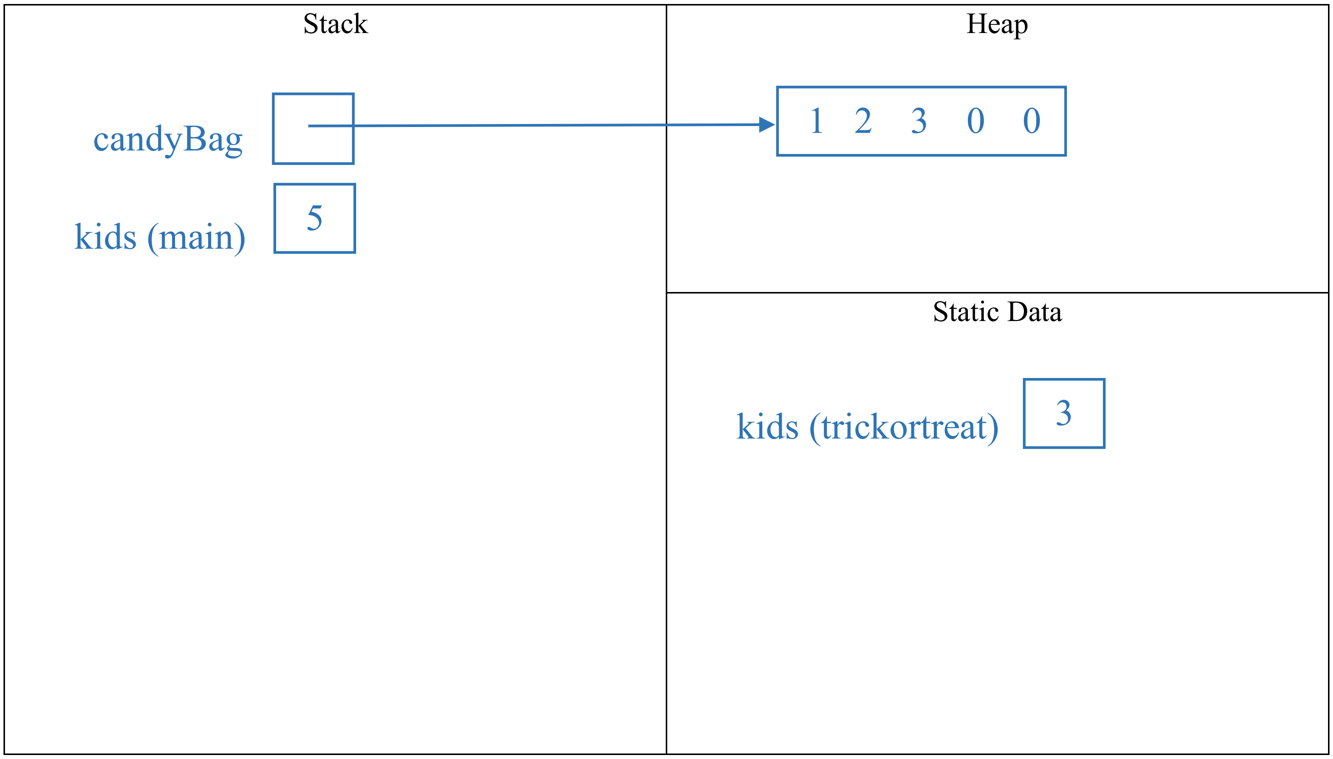
```
#include "trickortreat.h"
#define BAG_CAPACITY  5
#define NUM_PIECES    3

void InitializeCandy(int a[]) {
  for (int i = 0; i < BAG_CAPACITY; i++) {
    a[i] = EATEN_CANDY;
  }
}
int main(int argc, char *argv[]) {
  int *candyBag = (int*)malloc(BAG_CAPACITY * sizeof(int));
  int kids = 5;
  InitializeCandy(candyBag);
  for (int i = 0; i < NUM_PIECES; i++) {
    candyBag[i] = Dispense();
  }
  // *** HERE ***
  free(candyBag);
  return 0;
}
```

(A) [8 pts] Below, write the contents of `trickortreat.c` after it has been pre-processed.

```
int Dispense();
static int kids = 0;
int Dispense() {
  int candy =
    ((kids % 3) + 1);
  kids++;
  return candy;
}
```

(B) [20 pts] Draw a memory diagram showing the state of the program at "`*** HERE ***`". For your convenience, our two .c files are reprinted below.



(reprinted code below)

main.c

```
#include "trickortreat.h"
#define BAG_CAPACITY   5
#define NUM_PIECES     3

void InitializeCandy(int a[]) {
  for (int i = 0; i < BAG_CAPACITY; i++) {
    a[i] = EATEN_CANDY;
  }
}
int main(int argc, char *argv[]) {
  int *candyBag = (int*)malloc(
    BAG_CAPACITY * sizeof(int));
  int kids = 5;
  InitializeCandy(candyBag);
  for (int i = 0; i < NUM_PIECES; i++) {
    candyBag[i] = Dispense();
  }
  // *** HERE ***
  free(candyBag);
  return 0;
}
```

trickortreat.c

```
#include "trickortreat.h"
#define NUM_CANDY_TYPES   3
#define TO_CANDY(c)      ((c) + 1)

static int kids = 0;

int Dispense() {
  int candy =
    TO_CANDY(kids % NUM_CANDY_TYPES);
  kids++;
  return candy;
}
```

Question 6:

Consider the following C++ program:

```
void embiggen(int a[], int size) {
  for (int i = 0; i < size; ++i) {
    a[i] *= 10;
  }
}

int main(int argc, const char *argv[]) {
  int arr[] = {0, 1, 2, 3};

  int i = arr[0];
  i += 3;

  int &r = arr[1];
  r += 2;

  int *p = &(arr[2]);
  p += 1;

  embiggen(arr, 4);

  // *** HERE ***

  return 0;
}
```

[19 pts] When this program reaches “*** HERE ***”, what do each of these expressions evaluate to?

`i`	3
`r`	30
`*p`	30
`arr`	{ 0 , 30 , 20 , 30 }
`&i == &(arr[0])`	True [✓ False]
`&r == &(arr[1])`	[✓ True] False
`&r == &(arr[3])`	True [✓ False]
`p == &(arr[2])`	True [✓ False]
`p == &(arr[3])`	[✓ True] False

Question 7:

Our templated “Smart Vector” class stores pointers to dynamically-allocated objects and releases their memory when it goes out of scope. Furthermore, it implements “deep copy” semantics by copying the *pointees* rather than the pointers (ie, copying raw memory addresses) whenever a SmartVector is copied.

SmartVector.h

```
#ifndef SMARTVECTOR_H_
#define SMARTVECTOR_H_

extern const int kMaxSize;

template <typename T> class SmartVector {
 public:
  SmartVector() : currentSize_(0) { }
  SmartVector(const SmartVector &other) {
    // Implement me in Part (A)!
  }
  ~SmartVector() {
    for (int i = 0; i < currentSize_; ++i) {
      delete contents_[i];
    }
  }
  void Append(T *elt) {
    Verify333(currentSize_ < kMaxSize);
    contents_[currentSize_] = elt;
    currentSize_++;
  }
  T* Get(int idx) const {
    Verify333(idx >= 0 && idx < currentSize_);
    return contents_[idx];
  }
 private:
  T* contents_[kMaxSize];
  int currentSize_;
};
#endif  // SMARTVECTOR_H
```

SmartVector.cc

```
#include “SmartVector.h”

const int kMaxSize = 64;
```

(A) [10 pts] Implement SmartVector’s copy constructor.

```
SmartVector(const SmartVector &other) {
  currentSize_ = other.currentSize_;
  for (int i = 0; i < other.currentSize_; i++) {
    contents_[i] = new T( *(other.contents_[i]) );
  }

}
```

(B) [4 pts] SmartVector currently works on any T. Based on your new copy constructor, what restrictions now apply to T’s functionality? If there are changes, describe them below.

X There Are New Restrictions (described below) □ No New Restrictions

T needs to support copy-construction.

(C) [8 pts] Considering all we know about classes and deep copies, what is `SmartVector` missing and why does it matter?

SmartVector doesn’t comply with the “Rule of 3”; it needs to implement an assignment operator to avoid making shallow copies of its contained pointers.

If a SmartVector “b” is assigned to a SmartVector “a”, then both of them will attempt to delete the same contents when they go out of scope. This will result in a double-delete.

(D) [3 pts] Using 3 lines or fewer, write code that demonstrates the missing functionality discussed in (C). We’ve given you some starter code.

```
#include “SmartVector.h”

int main(int argc, const char *argv[]) {
  SmartVector<int> v1;
  v1.Append(new int(351));
  v1.Append(new int(333));

  __ SmartVector<int> v2; ___________

  __ v2 = v1; ______________________

  __________________________________
  return 0;
}
```

Question 8:

[1 pt; all non-empty answers receive this point] Select one member of the course staff. Describe or draw an emoji representing that person.

Congratulations on finishing the midterm!