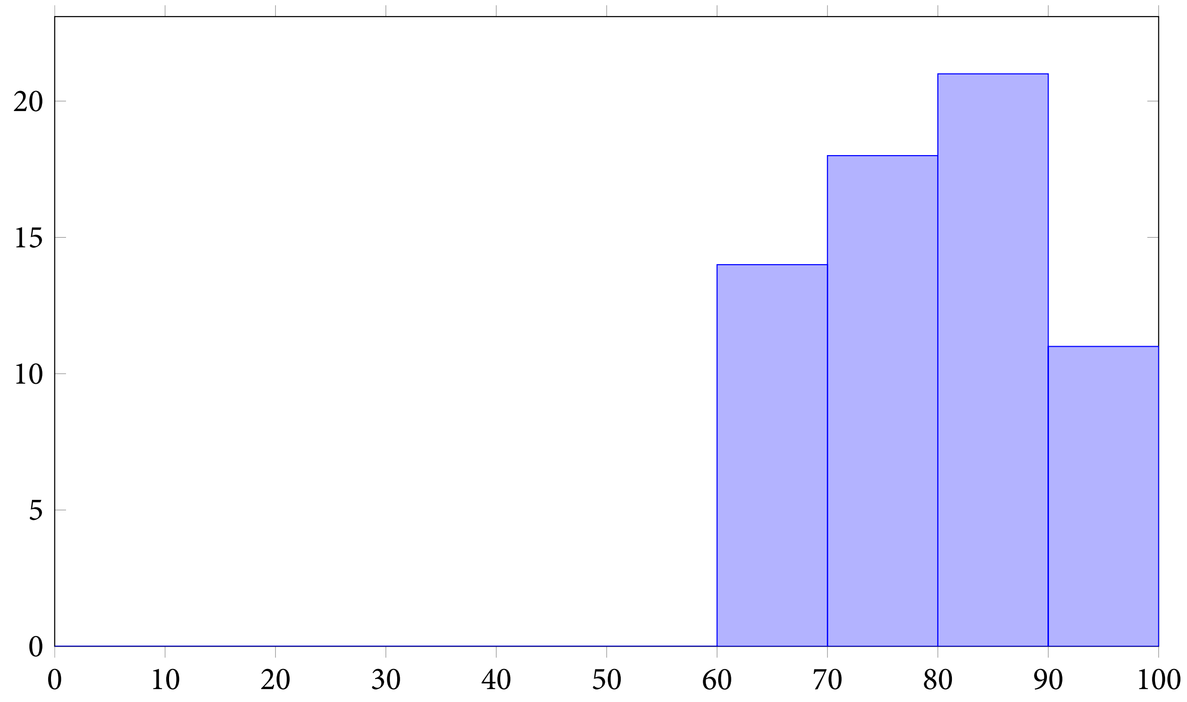
COMPUTER SCIENCE & ENGINEERING
UNIVERSITY *of* WASHINGTON

CSE 333 Winter 2015
Midterm Solutions



mean 78.61, median 79, stdev 10.19

I. Integers in the Machine City

Alyssa P. Hacker wants to exchange data with Ben Bitdiddle over network. They plan to use the External Data Representation (XDR) protocol as defined in RFC 4506.

(a) (12 points) An XDR unsigned integer encodes a 32-bit non-negative integer in the range $[0, 2^{32} - 1]$ in big endian. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively.

(MSB)			(LSB)
byte 0	byte 1	byte 2	byte 3

Please help Alyssa complete the following C function that encodes an XDR unsigned integer. For example, when Alyssa uses the function to encode 3735928559 (hex value `0xdeadbeef`), the resulting bytes 0–3 (in hex) should be `0xde`, `0xad`, `0xbe`, and `0xef`, respectively.

```c
#include <stdint.h>

void xdr_encode_uint(uint32_t x, uint8_t buf[4]) {
  /* byte 0 */
  buf[0] =

  /* byte 1 */
  buf[1] =

  /* byte 2 */
  buf[2] =

  /* byte 3 */
  buf[3] =
}
```

Solution:

```c
buf[0] = (x >> 24) & 0xff;
buf[1] = (x >> 16) & 0xff;
buf[2] = (x >> 8) & 0xff;
buf[3] = x & 0xff;
```

It's okay to omit "`& 0xff`" in all the above cases.

(b) (8 points) Ben Bitdiddle owns a PlayStation 3 game console, which uses the Cell processor in big-endian mode. Ben has installed Linux on the game console, and decides to run following C program there, using a 32-bit unsigned integer received from Alyssa.

```c
#include <inttypes.h>
#include <stdio.h>
#include <string.h>

int main(void) {
  uint32_t x = 0xdeadbeef;
  uint16_t a, b;

  a = (uint16_t)x;
  memcpy(&b, &x, sizeof(b));
  printf("%" PRIx16 " ", a);
  printf("%" PRIx16 "\n", b);
  return 0;
}
```

- `memcpy(dst, src, n)` copies `n` bytes from memory area `src` to memory area `dst`.
- `PRIx16` is a format specifier for 16-bit lower-case hex.

Which of the following should Ben see from the output of the program? Check the appropriate box (no need to justify your answers here).

☐ `dead beef` [✓ **`beef dead`**] ☐ `dead dead` ☐ `beef beef`
☐ `adde efbe` ☐ `efbe adde` ☐ `adde adde` ☐ `efbe efbe`

II. The eternal war in memory

Ben Bitdiddle is implementing the linked list from Homework #1. Below is part of his code.

```c
typedef uint32_t HWSize_t;
typedef void *LLPayload_t;

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;

typedef struct {
  HWSize_t       num_elements;  // # elements in the list
  LinkedListNode *head;         // head of list, or NULL if empty
  LinkedListNode *tail;         // tail of list, or NULL if empty
} *LinkedList;

bool PushLinkedList(LinkedList list, LLPayload_t payload) {
  // defensive programming: check argument for safety.
  Verify333(list != NULL);

  // create a new node
  LinkedListNode new_node;
  LinkedListNode *ln = &new_node;

  // set the payload.
  ln->payload = payload;
  ln->next = list->head;
  ln->prev = NULL;

  ...
  list->head = ln;
  list->num_elements++;

  // return success
  return true;
}
```

See next page for questions.

Ben is using `attu.cs.washington.edu`, which is running x86_64 Linux. Therefore, consider the following questions for x86_64 Linux only.

(a) (10 points) Circle true or false for each statement (no need to justify your answers here).

True **False** The value of `sizeof(HWSize_t)` is 32.

Solution: F - should be 4

True **False** `sizeof(struct ll_node *)` is equal to `sizeof(LinkedListNode)`.

Solution: F

True **False** `LLPayload_t` is a pointer type.

Solution: T

True **False** `LinkedListNode` is a pointer type.

Solution: F

True **False** `LinkedList` is a pointer type.

Solution: T

(b) (10 points) Ben's `test_suite` keeps crashing. Please help him fix the problem. Describe the code you would like to add and/or remove. Hint: you only need to change two lines.

Solution: Replace lines 21 and 22 with the following:

```
LinkedListNode *ln = malloc(sizeof(LinkedListNode));
if (!ln) return false;
```

III. I/O

(a) (10 points) Circle true or false for each statement (no need to justify your answers here).

True **False** `fopen` returns a pointer of type `FILE *`; to get the corresponding low-level file descriptor, cast the pointer to an `int`.

Solution: F - casting an `FILE *` to `int` won't work; one can use `fileno` to get the corresponding file descriptor

True **False** The first `fread(buf, 1, 100, fd)` from a 1000-byte long file would always return 100.

Solution: F - the `fread` call returns less than 100 if a read error occurs

True **False** `fread` is faster than `read` when the program performs multiple one-byte reads.

Solution: T - `fread` is buffered

True **False** `fwrite` can achieve the same guarantee as `write` in an event of power failure.

Solution: F - `fwrite` is buffered

True **False** After invoking `close(fd)`, any use of the same file descriptor `fd`, such as `read(fd, ...)`, will cause a memory corruption.

Solution: F - `read` will set `errno` to `EBADF` (rather than memory corruption)

(b) In 2009, a Ubuntu user reported a massive data loss when using KDE and the ext4 file system (Bug 317781): after a system crash and reboot, "pretty much any file written to by any application" became zero bytes! This sparked a long discussion on who to blame.

Alyssa P. Hacker is interested in this issue. Using `strace` she observes that these applications produce two common patterns of system call sequences when they update an existing file.

Pattern 1: truncate an existing file to zero bytes (i.e., `open` with `O_TRUNC`) and write new data to that file.

```
fd = open("file", O_TRUNC);
write(fd, data);
close(fd);
```

Pattern 2: create a temporary file, write new data to the temporary file, and rename the temporary file to replace an existing file.

```
fd = open("file.tmp", O_CREAT, ...);
write(fd, data);
close(fd);
rename("file.tmp", "file");
```

Assume the hardware (e.g., CPU, memory, and disk) works correctly.

i. (5 points) For pattern 1, do you think it is possible for an existing `"file"` to become zero bytes when the computer loses power during the system calls? Explain why or why not.

Solution: Yes, for example, if the computer loses power after `open` and before write.

ii. (10 points) For pattern 2, do you think it is possible for an existing `"file"` to become zero bytes when the computer loses power during the system calls? Explain why or why not.

Solution: Yes, for example, if the computer loses power after `rename` and before the kernel flushes the content of the new `"file"` to disk. To fix this, one needs to add `fsync(fd)` before `close(fd)`.

We also accept "no" if the answer mentions that the latest version of the ext4 file system would flush the file upon `rename`.

IV. C++ minus minus

Consider a base class Dog and a derived class Husky below. There are lots of empty spaces where perhaps things are missing.

```
#include <stdio.h>

class Dog {
 public:
  int eat() { return printf("Dog::eat\n"); }
  virtual void bark() { printf("Dog::bark\n"); }
  virtual ~Dog() = default;
};

class Husky : public Dog {
 public:
  // constructor
  Husky(________ double ________ weight) ________ : weight_(weight) {}

  // return weight
  double _______ getWeight() ______ { return weight_; }

  int eat() { return printf("Husky::eat\n"); }
  _______ _______ bark() _______ { printf("Husky::bark\n"); }
  virtual void mascot() { printf("Husky::mascot\n"); }

 private:
  double weight_;
};
```

(a) (8 points) Complete the declarations by filling in any necessary keywords or symbols. You should leave each space empty if that is appropriate, or write in some combination of `&`, `*`, `const`, `static`, `void`, `virtual`, `override`, or whatever else is needed to declare things correctly. If something is optional or if you have choices between more than one way to fill in a blank, make the most appropriate choice.

Solution:

```
Husky(double weight) : weight_(weight) {}
double getWeight() const { return weight_; }
virtual void bark() override { printf("Husky::bark\n"); }
```

It's okay to omit `virtual` or `override` for `bark`.

(b) (16 points) Finally, here's some setup code:

```
Husky husky(40.0);
Dog *pDog = &husky;
Dog &dog = *pDog;
```

Below is a list of method invocations. For each, indicate whether 1) it causes a compile-time error, 2) it invokes the method in class `Dog`, or 3) it invokes the method in class `Husky`.

`dog.eat();`	☐ 1	■ **2**	☐ 3
`dog.bark();`	☐ 1	☐ 2	■ **3**
`husky.eat();`	☐ 1	☐ 2	■ **3**
`husky.bark();`	☐ 1	☐ 2	■ **3**
`husky.mascot();`	☐ 1	☐ 2	■ **3**
`pDog->eat();`	☐ 1	■ **2**	☐ 3
`pDog->bark();`	☐ 1	☐ 2	■ **3**
`pDog->mascot();`	■ **1**	☐ 2	☐ 3

(c) (1 point) What header file(s) do you need to include if you want to use `std::cout` in C++ instead of `printf`?

Solution:

`#include <iostream>`

V. CSE 333

We'd like to hear your opinions. Any answer, except no answer, will receive full credit.

(a) (4 points) This year we introduced paper reading assignments. Did you find them useful? What should we do to improve them?

> **Solution:** too long & time consuming (19), too difficult & technical (8), need more details (5), not useful (4), need more specific questions (4), need more readings (4)

(b) (3 points) What is the best aspect of CSE 333?

> **Solution:** low-level (22), exercises & homework (21), C/C++ (16), staff (7), practical (6)

(c) (3 points) What is the worst aspect of CSE 333?

> **Solution:** too much work (10), bugs, leaks & seg faults (6), exercises (6), exam (5), homework (5)

End of Quiz