CSci 2021 Section 010
Fall 2018
Midterm Exam 1 (solutions)
October 8th, 2018
Time Limit: 50 minutes, 3:35pm-4:25pm

- This exam contains 6 pages (including this cover page) and 4 questions. Once we tell you to start, please check that no pages are missing.
- Before starting the exam, you can fill out your name and other information of this page, but don't open the exam until you are directed to start. Don't put any of your answers on this page.
- You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic devices: no calculators, smart phones, laptops, etc.
- You may ask clarifying questions of the instructor or TAs, but no communication with other students is allowed during the exam.
- Please read all questions carefully before answering them. Remember that we can only grade what you write on the exam, so it's in your interest to show your work and explain your thinking.
- Students often find that the questions vary in difficulty. Your best strategy is usually to skim over all the questions, and then start working on the ones that look easiest. We also suggest that you leave time at the end to attempt every question, since we can't give you any partial credit if you leave a question blank.
- By signing below you certify that you agree to follow the rules of the exam, not to share exam material with other students before their exams, and that the answers on this exam are your own work only.

The exam will end promptly at 4:25pm. Good luck!

Your name (print): _____

Your UMN email/X.500: _____`@umn.edu`

Sign and date: _____

Question	Points	Score
1	20	
2	31	
3	24	
4	25	
Total:	100	

1. (20 points) C declarations and uses.

   In the left column are ten declarations of a C variable named `x`. In the right column are ten uses of a variable named `x`. However, the way you use a variable depends on its type: most of these uses would be illegal or nonsensical for most of the types. Match each declaration on the left with the legal and most sensible use on the right by writing the letter of the use on the blank. Each choice will be used exactly once.

   The structure `pair` is declared as `struct pair {int a; int b;};`. It is OK if the variable would need to be initialized in between the declaration and the use, even if we haven't shown that. `sinf` is an implementation of the sine function from trigonometry.

(a) __**J**__ `int x[10];`

(b) __**G**__ `int x;`

(c) __**C**__ `struct pair *x[20];`

(d) __**E**__ `struct pair *x;`

(e) __**D**__ `char x;`

(f) __**I**__ `char *x;`

(g) __**H**__ `float x;`

(h) __**B**__ `int *x;`

(i) __**F**__ `struct pair x[100];`

(j) __**A**__ `struct pair x;`

A. `x.a = 0;`

B. `x = malloc(20 * sizeof(int));`

C. `x[10]->b = x[9]->b;`

D. `x = '*';`

E. `x->b++;`

F. `x[12].a = 0;`

G. `x <<= 20;`

H. `x = sinf(x); /* sine func. */`

I. `x = strdup("message");`

J. `int num_elements = sizeof(x)/sizeof(int);`

Integer arrays and integer pointers are similar for many purposes, but only a pointer can be initialized with `malloc` *(B), and* `sizeof` *gives a useful result only for an array (J). Similarly, though there was no char array possibility, a string copied by* `strdup` *could only go in a character pointer.*

Characters are a sub-range of integers, but character constants are typically stored in characters (D), whereas a shift amount of 20 would only make sense for an integer (G).

A float can be converted to an integer, but since the sine function returns a value between -1 and 1, you would usually just get 0 if you stored it in an integer (H).

The four structure based types all end with accesses to fields, but you only use indexing brackets when there is an array involved (C, F), and you only use -> when there is a pointer (C, E).

2. (31 points) Subtraction and overflow.

   Below are the operation tables for unsigned and two's complement subtraction of 3-bit values. An entry in each table shows the result you get if you subtract the value in the column label from the value in the row label. For instance, the number **2** shown in boldface represents that if you perform an unsigned subtraction 3 minus 1, you get 2 (011 - 001 = 010). Some of the entries in the tables are circled, indicating that the results demonstrate unsigned (first table) or signed (second table) overflow. Recall that we say a result has overflowed if it is different from the result that would be computed using unlimited-range mathematical integers.

   We have left some of the table entries blank: your job is to fill them in correctly. You should circle some of the entries you write, according to the same overflow rules. But you shouldn't circle any of the numbers we have written.

   Recommendation: because there are a lot of entries for you to fill in, you probably don't want to do each calculation individually in binary. Instead, do some operations carefully until you see patterns that can let you fill in answers more quickly.

   3-bit unsigned subtraction, with unsigned overflow circled:

$r -_3^u c$	0	1	2	3	4	5	6	7
0	0	⑦	⑥	⑤	④	③	②	①
1	1	0	⑦	⑥	⑤	④	③	②
2	2	1	0	⑦	⑥	⑤	④	③
3	3	**2**	1	0	⑦	⑥	⑤	④
4	4	3	2	1	0	⑦	⑥	⑤
5	5	4	3	2	1	0	⑦	⑥
6	6	5	4	3	2	1	0	⑦
7	7	6	5	4	3	2	1	0

   Some basic patterns that work with both tables have to do with how adjacent table entries are related. Moving one step left is adding 1 to the number being subtracted, so it makes the result one less (except with overflow, but the overflow behavior is the same as subtracting 1). Moving one step down is adding 1 to the number being subtracted from, so it makes the result one more. Moving one step left and one step down (i.e., one step diagonally toward the lower right), the two changes cancel out and leave the result the same. That's why all the entries along a downward-sloping diagonal are the same. Another thing to check is that because modular arithmetic forms a group, all the possible results appear exactly once in every row and in every column.

   In the unsigned table, all the entries above the 0 diagonal should be negative. But of course no negative numbers can be represented in unsigned, so they are all overflows.

3-bit two's complement subtraction, with signed overflow circled:

$r -_3^t c$	-4	-3	-2	-1	0	1	2	3
-4	0	-1	-2	-3	-4	(3)	(2)	(1)
-3	1	0	-1	-2	-3	-4	(3)	(2)
-2	2	1	0	-1	-2	-3	-4	(3)
-1	3	2	1	0	-1	-2	-3	-4
0	(-4)	3	2	1	0	-1	-2	-3
1	(-3)	(-4)	3	2	1	0	-1	-2
2	(-2)	(-3)	(-4)	3	2	1	0	-1
3	(-1)	(-2)	(-3)	(-4)	3	2	1	0

The same basic patterns for filling in table entries work in the two's complement table too. The overflow happens in different places, but it looks somewhat similar in that it happens as you cross a diagonal line. In this table we've drawn in only the middle lines to show how the table is divided into four quadrants based on the signs of the two input values. If the two inputs have the same sign, no overflow is possible, just like you can't get signed overflow by adding two values of opposite sign. The first column, which represents subtracting -4, is a special case if you try to think about it as negating -4 followed by addition, because the negation 4 is not representable. But it fits in a uniform pattern if you just think about the subtraction inputs and outputs, as we do here. Overflow is only possible when the two inputs have different signs: in this case the difference should have the same sign as the left operand. If it has the opposite sign instead, that's overflow.

3. (24 points) Matching floating point.

   On the left are descriptions of 8 floating-point numbers. On the right are bit patterns for IEEE single precision floating-point numbers, divided up into groups as the sign bit, the exponent bits, and the fraction bits. Match each description on the left to a bit pattern on the right by filling in the corresponding letter. Each bit pattern is used exactly once.

   This question should not require detailed computations. Instead, try looking for differences between the numbers that lead to differences in their floating-point representations, starting with the easiest (for instance, most distinctive) values first.

(a) __**C**__	10^{-44}	A. 0 11111111 00000000000000000000000
(b) __**H**__	10^{30}	B. 1 00000000 00000000000000000000000
(c) __**G**__	$\pi \approx 3.14159$	C. 0 00000000 00000000000000000000111
(d) __**A**__	$+\infty$	D. 1 10000000 10000000000000000000000
(e) __**D**__	-3	E. 0 10000000 01011011111100001010100
(f) __**F**__	2^{100}	F. 0 11100011 00000000000000000000000
(g) __**E**__	$e \approx 2.71828$	G. 0 10000000 10010010000111111011011
(h) __**B**__	-0	H. 0 11100010 10010011111001011001010

Only zero and negative zero have all of their exponent and fraction bits 0. B has the sign bit set, so it must be negative zero.

All ones in the exponent bits and all zeros in the fraction make an infinity, and A is positive, so it's $+\infty$.

A value with an all zero bits exponent but non-zero fraction bits, C, is a denormalized number that is very close to zero. The only number here that is positive but has very small finite magnitude is 10^{-44}.

The only remaining negative number D must be -3. It is useful to remember for later here that the exponent pattern 10000000 means 2^1*, and* 3 *is* $1.1_2 \cdot 2^1$.

e *and* π *are both between 2 and 4, and have same exponent as -3, but are positive, so they must be E and G in some order.* π *of course is bigger than* e*, and because it's bigger than 3, the first digit of its fraction bits is* 1*. On the other hand* e *is less than 3, so its first fraction bit must be* 0*. Thus E is* e *and G is* pi.

This leaves two large but finite numbers F and H, which must be 10^{30} *and* 2^{100} *in some order. The easiest way to differentiate them is that* 2^{100} *is a power of two, so all of its fraction bits must be zero, while* 10^{30} *is not. Thus F is* 2^{100} *and H is* 10^{30}*. You may also remember that* 2^{10} *is 1024 and* 10^3 *is 1000, so* $2^{100} = (1024)^{10}$ *and* $10^{30} = (1000)^{10}$*, so* 2^{100} *is slightly bigger.*

4. (25 points) Pointers.

   Below is some C code that uses pointers in a variety of ways. In the blank next to each call to `printf`, write the number that would be printed. If you want, you may find it helpful to use the space on the right to make notes on what variables contain at each point, or to draw diagrams of what is pointing to what.

```
int a = 10; int b = 20; int c = 30; int d = 40; int e = 50;

int *p1 = &a;
p1 = 0; /* setting the pointer to null has no effect on a */

printf("%d\n", a);          ___ 10 ___

int *p2 = &b;
*p2 += 5; /* this increases b from 20 to 25 */

printf("%d\n", *p2);        ___ 25 ___

int *p3 = &b;
int *p4 = &c;
int *p5 = p4; /* p5 now points to c */
p4 = p3;      /* p4 now points at b */
p3 = p5;      /* p3 now points at c*/
p4 += 10;     /* p4 now points somewhere weird */

printf("%d\n", *p3);        ___ 30 ___

int *p6 = &d;
int **pp7 = &p6; /* *pp7 is p6, so **p7 is d */
if (pp7)         /* pp7 is non-null, so true */
    (*p6)++;     /* increases d from 40 to 41 */
else
    *(p6++);     /* not sensible, but not executed  */

printf("%d\n", **pp7);      ___ 41 ___

int **pp8 = malloc(3 * sizeof(int *));
pp8[0] = &c; /* *pp8 also points at c */
pp8[1] = &d;
pp8[2] = &e;
int ***ppp9 = &pp8;
***ppp9 += ***ppp9; /* increases c from 30 to 60 */
pp8[1] = pp8[0];    /* pp8[1] now points at d */
pp8[0] = pp8[2];    /* pp8[0] and *pp8 now point at e */
pp8[2] = pp8[1];    /* pp8[2] now points at d */

printf("%d\n", ***ppp9 >> 1); /* e / 2 */    ___ 25 ___
```