15-122: Principles of Imperative Computation, Spring 2023

Written Homework 11

Due on Gradescope: Monday 10th April, 2023 by 9pm

Name: _____

Andrew ID: _____

Section: _____

This written homework provides practice with C features such as arrays, undefined behaviors, and stack allocation.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- ***PDFescape*** or ***DocHub***, two web-based PDF editors that work from anywhere.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though*.

Caution Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

Question:	1	2	3	4	Total
Points:	3.5	2.5	4	5	15
Score:					

1. **Accessing Data on the Stack**

> *In this question, we assume the usual 2's complement implementation of unsigned and signed* ***char*** *(8 bits, one byte),* ***short*** *(16 bits, two bytes), and* ***int*** *(32 bits, four bytes). Pointers are 64 bits, eight bytes. We also assume that numbers are represented in memory with their most significant bits at lower addresses.*

During execution, the local variables of a function are stored on the **system stack**. In particular, so are stack-allocated arrays and structs. Each item on the stack starts at a unique address, but each address contains exactly one byte of data. Therefore the value of a variable will often span multiple bytes.

Assume the function being executed contains the following local declarations:

```
// a stack-allocated array
char cs[] = {'a', 'b', 'c', 'd'};
// an int
int i = 0x01ABCDEF;
```

The compiler decides on the layout of their component bytes onto the stack. Assume it arranged them as shown on the right of this text: the variable `cs` starts at address `0xC4` which contains `0x61` (the ASCII value of `'a'`), the variable `i` starts at address `0xC8` where it stores the 8 most significant bits of its value.

Notice that while `cs` and `i` look similar on the stack, `cs` is an array — its values are read by indexing — whereas `i` is an integer with value `0x01ABCDEF`.

Var.	Addr.	Stack
	...	...
	0xCB	EF
	0xCA	CD
	0xC9	AB
i	0xC8	01
	0xC7	64
	0xC6	63
	0xC5	62
cs	0xC4	61

1pt **1.1** For the following program, draw the contents of the stack after the code shown is executed. Assume the compiler arranges the bytes as in the previous example. Leave any byte that does not contain program data blank.

```
char C[] = {'c', 'a', 't', '\0'};
int n = 0x00C0FFEE;
```

What is the value of `&n`? _____

Fill in with the correct type: _____ `x = &n;`

Var.	Addr.	Stack
	...	...
	0x87	
	0x86	
	0x85	
n	0x84	
	0x83	
	0x82	
	0x81	
C	0x80	

1pt **1.2** Assume we are given the function

```
bool parse(char *x, int *p);  // Returns true iff parse succeeds
```

which reads the string representation `x` of a decimal number (e.g., `"15122"`) and stores the corresponding integer (here `15122`) at `p`. If `x` is not a decimal number (e.g., `"hello!"`), it returns `false` — it returns `true` to signal a successful parse.

Now we want to implement the function `test_parse` which calls `parse` on its input and prints whether the parse was successful or not. Before we knew about getting the address of a variable, we would have implemented `test_parse` as follows:

```
void test_parse(char *x) {
  REQUIRES(x != NULL);
  int *i = xmalloc(sizeof(int));
  if (parse(x, i)) printf("Success: %d!\n", *i);
  else printf("Failure.\n");
  free(i);
}
```

In particular, we had to allocate space for the variable `i` so that `parse` had somewhere to write the result of a successful parse.

But now that we know how to get the address of a variable, we can reimplement `test_parse` so that it does not heap-allocate `i` (and risk leaking memory). Complete the following code:

```
void test_parse_lean(char *x)
  REQUIRES(x != NULL);
  int i;




}
```

1pt **1.3** Assume the string `"410"`, which resides at address `0xA7`, is passed into `test_parse_lean` so that, just before the call to `parse`, the contents of the stack are displayed to the right of this text. For succinctness, we wrote the memory address of the string `"410"` as `0xA7` on the stack drawing — an actual address would span 8-bytes of stack space.

Var.	Addr.	Stack
	...	0xA7
x	0x80	
	0x7F	00
	0x7E	00
	0x7D	00
i	0x7C	00

What are the hex values of the arguments passed to `parse`? *Hint: recall what the values of pointer types are.*

`parse(_____, _____)`

What are the contents of the stack when `parse` has finished executing? Write all values in hex in the diagram to the right of this text.

Var.	Addr.	Stack
	...	
	0x80	0xA7
	0x7F	____
	0x7E	____
	0x7D	____
i	0x7C	____

0.5pts **1.4** One of the dangers of using the address-of operator is that stack addresses get reused. Consider a version of `test_parse_lean` that returns the address of the parsed value.

```
int *test_parse_weird(char *x) {
  REQUIRES(x != NULL);
  int i;
  // Your code from test_parse_lean goes here
  return &i;
}

int main() {
  int *n1 = test_parse_weird("122");
  int *n2 = test_parse_weird("150");
  ASSERT(*n1 != *n2);   // we expect n1 and n2 to differ
  return 0;
}
```

Since we don't know anything about where `i` appears on the stack, both calls to `test_parse_weird` could end up putting `i` at the same address (i.e., the values would be overwritten). Answer the following questions under this assumption.

If `n1 == 0xB0`, the value of `n2` is _____

The argument of `ASSERT` evaluates to _____ (`true` or `false`?)

2. **Undefined Behavior**

   C is difficult to program in because it is so permissive. Many things which are errors in C0 are undefined in C, so the behavior can change even between runs.

   Consider the following program, where we assume **`int`**s are 4 bytes long and pointers are 8 bytes long:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5    int *n = malloc(2 * sizeof(int));
6    *n = 15122;
7
8    int *A;
9    *A = 6;
10
11   if (*n == 15122) printf("Yay!\n");
12   else printf("Blasphemy\n");
13
14   free(n);
15   return 0;
16 }
```

   Assume that when the execution reaches line 7, the heap looks as follows:

   The grayed out region is allocated by you on line 5 — i.e., `malloc` returned `0xA4`. Access to the red region is forbidden (doing so will result in a segmentation fault). Although executing line 9 is undefined behavior, you may assume it succeeds unless it writes to the red region.

1.5pts

2.1 Since local variables are not automatically initialized, `A` could be anything. For each of the following possible values of `A`, state whether the program runs normally, prints `"blasphemy"`, or throws an error. Mark only one outcome for each row.

`A == 0xA0`:	◯ prints `"Yay!"`	◯ prints `"Blasphemy"`	◯ segfaults
`A == 0xA4`:	◯ prints `"Yay!"`	◯ prints `"Blasphemy"`	◯ segfaults
`A == 0xA6`:	◯ prints `"Yay!"`	◯ prints `"Blasphemy"`	◯ segfaults
`A == 0xA8`:	◯ prints `"Yay!"`	◯ prints `"Blasphemy"`	◯ segfaults
`A == 0xAC`:	◯ prints `"Yay!"`	◯ prints `"Blasphemy"`	◯ segfaults
`A == 0xAE`:	◯ prints `"Yay!"`	◯ prints `"Blasphemy"`	◯ segfaults

For the next two task, you will need to compile this code and run it with Valgrind. You will need to use the following compilation command:

```
# gcc -g file.c
```

(without additional flags) where `file.c` is the name of the file where you saved this code. Read the guide to success on **How to use Valgrind** on Piazza to interpret the output of Valgrind.

0.5pts

2.2 Valgrind will point you to the problem with this code *even if the program runs correctly*. What is the problem in the code, and what is the corresponding error message that Valgrind displays?

Valgrind error message: _____

because _____

0.5pts

2.3 Using this error message, fix the error in `file.c`. Then, change line 14 to `free(n+1)`. What is the *exact error message* that Valgrind displays now? Explain your answer.

Valgrind error message: _____

because _____

Always use Valgrind!

3. **Strings in C**

Unlike C0, C does not have a **`string`** type. Rather, strings in C are arrays of **`char`**'s that end in the `NUL` character, the character with ASCII value 0, which we write as `'\0'`. They can live in three places:

1. In the **DATA** segment of memory. Space is automatically created for them, it is undefined behavior to write to them (they are "read-only"), and they are automatically `NUL`-terminated. You create them when you write string literals, such as `"functions are pointers"`, in your code.
2. On the **heap**. Space needs to be provided for them using `malloc` or similar, and it needs to be freed.
3. On the **stack**, as stack allocated arrays. Space for them is created when entering a function and disappears when returning from it.

As you might anticipate, strings can lead to a number of errors! The tasks in this question will explore the ways things can go wrong.

1.5pts **3.1** The following function writes `A` to `res`, followed by these same characters in reverse. For example, if `A` is `"honk!"`, then `res` will be `"honk!!knoh"`.

```
1  void mirror(char *A, char *res) {
2    REQUIRES(A != NULL);
3    REQUIRES(res != NULL);
4
5    int n = strlen(A);
6    for (int i = 0; i < n; i++) {
7      res[i] = A[i];
8      res[n+i] = A[n-i-1];
9    }
10   res[2*n] = '\0';
11 }
```

Depending on what string is passed to `mirror` as `A` and what we do with our memory, we could run into a number of common issues:

- We could read a value out of bounds or read from memory we have already freed. These are *invalid reads*.
- We could write to a value out of bounds, write to memory we have already freed, or write to read-only memory. These are *invalid writes*.
- We could free a value we have already freed or free a pointer we did not allocate. These are *invalid frees*.

Here are some test cases using `mirror`. Determine whether they could result in an invalid read, an invalid write, or an invalid free. For each one, check all boxes that apply. If none do, check "ALL GOOD". Remember to account for the `NUL`-terminator!

When determining the effects of each statement, assume the previous ones (except the setup code before the first call to `mirror`) did not run and undefined behavior does not propagate to later function calls on the same line.

The library function `strncpy(s1,s2,n)` copies the first `n` characters of `s2` into `s1`.

```
int main() {
  char A[] = "hi!";
  char B[] = {'h', 'i','!'};
  char *C = "hi!";
  char *D = xmalloc(sizeof(char) * 4);
  strncpy(D, C, 4);
  char *E = D;

  char *a = xmalloc(sizeof(char) * 7);
  char b[7];
  char c[] = "hi";
  char *d = "goodbye";
  char *e = xmalloc(sizeof(char) * 12);

  //                                Invalid  Invalid  Invalid   ALL
  //                                 read     write    free    GOOD
  mirror(A, a);                      ☐        ☐        ☐       ☐
  mirror(B, b);                      ☐        ☐        ☐       ☐
  mirror(C, c);                      ☐        ☐        ☐       ☐
  mirror(D, d);                      ☐        ☐        ☐       ☐
  mirror(E, e);                      ☐        ☐        ☐       ☐
  mirror(A, b); free(b);             ☐        ☐        ☐       ☐
  mirror(C, e); free(e);             ☐        ☐        ☐       ☐
  free(d); mirror(E, c);             ☐        ☐        ☐       ☐
  free(a); mirror(B, d);             ☐        ☐        ☐       ☐
  free(E); mirror(D, a); free(D);    ☐        ☐        ☐       ☐
  return 0;
}
```

1pt

3.2 We can't fix all of these problems, but we can eliminate some by returning the result. Complete the code of `mirror2` so that it cannot perform invalid writes.

```
char *mirror2(char *A) {
  REQUIRES(A != NULL);

  int n = strlen(A);
  char *res = xcalloc(sizeof(char), _____________________________);









  return _____________________________________________________;
}
```

1.5pts

3.3 This version prevents invalid writes, but it introduces more possible leaks. For each line, mark it if it allocates memory that gets leaked when the program ends. Assume that all earlier lines ran but ignore any undefined behavior on them.

```
int main() {
  char A[] = "hi!";
  char B[] = {'h', 'i','!'};
  char *C = "hi!";
  char *D = xmalloc(sizeof(char) * 4);
  strncpy(D, C, 4);
  char *E = D;

  char *a, *b, *c, *d, *e;
```

	Invalid read	Invalid free	LEAK	ALL GOOD
`a = mirror2(A);`	☐	☐	☐	☐
`b = mirror2(B);`	☐	☐	☐	☐
`c = mirror2(C);`	☐	☐	☐	☐
`d = mirror2(D);`	☐	☐	☐	☐
`e = mirror2(E);`	☐	☐	☐	☐
`b = mirror2(A); free(b);`	☐	☐	☐	☐
`e = mirror2(C); free(d);`	☐	☐	☐	☐
`free(e); c = mirror2(E);`	☐	☐	☐	☐
`free(a); d = mirror2(B);`	☐	☐	☐	☐
`free(E); a = mirror2(D); free(D);`	☐	☐	☐	☐

```
  return 0;
}
```

4. Casting C Things

Since C has integers of different sizes, we often find it necessary to cast between them. In this question, we will make the following assumptions:

- whether signed or unsigned, a **`char`** has 8 bits (one byte), a **`short`** has 16 bits (two bytes), and an **`int`** has 32 bits (four bytes);
- signed numbers are implemented using two's complement (or equivalently the code was compiled with the `-fwrapv` flag).

2pts **4.1** Given the declarations

```
int sign_int = -7;
unsigned int un_int = 40125;
signed char sign_char = -114;
unsigned char un_char = 221;
```

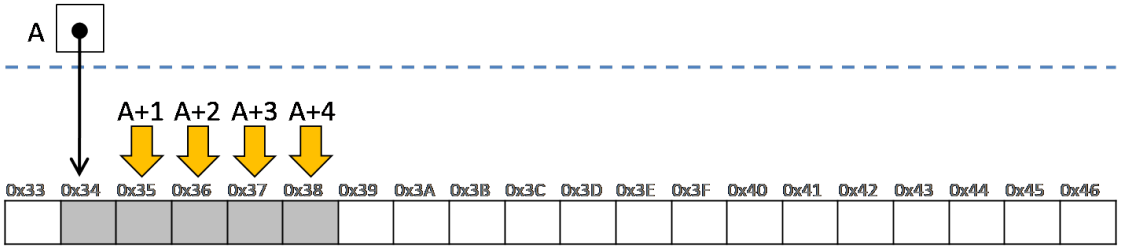
fill in the table below. In this task, fully simplify decimal values and always write the leading zeros in the hex representation, so that an **`int`** is 8 hex digits, a **`short`** 4, and a **`char`** 2. If the result of a cast is implementation-defined, assume the common rules seen in class but mark this result with **`(ID)`**.

	C expression	Decimal value	Hexadecimal
1.	`sign_int`	`-7`	`0xFFFFFFF9`
2.	`(unsigned int)sign_int`		
3.	`un_int`	`40125`	
4.	`(int)un_int`		
5.	`(unsigned short)un_int`		
6.	`(short)(unsigned short)un_int`		
7.	`sign_char`	`-114`	
8.	`(int)sign_char`		
9.	`(unsigned int)(int)sign_char`		
10.	`un_char`	`221`	
11.	`(signed char)un_char`		
12.	`(int)(signed char)un_char`		
13.	`(unsigned int)(int)(signed char)un_char`		
14.	`(unsigned int)un_char`		

1pt

4.2 An array's address is its starting byte. The type of an array tells the compiler how many bytes to read when accessing an element.

Consider the array `A` represented in memory by the shaded cell in the following memory diagram:



What are the possible types of the elements of `A`? Mark all that apply.

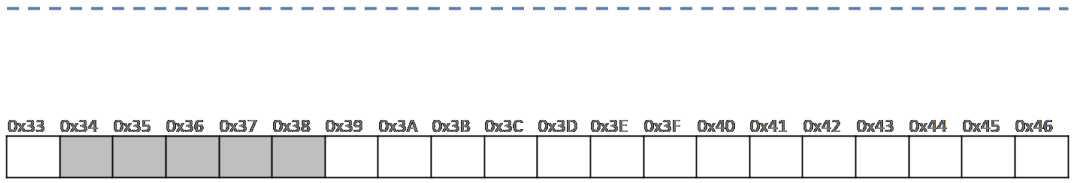
☐ **`char`**	☐ **`unsigned char`**	☐ **`short`**	☐ **`unsigned short`**
☐ **`int`**	☐ **`unsigned int`**	☐ **`long`**	☐ **`unsigned long`**
☐ `int8_t`	☐ `int16_t`	☐ `int32_t`	☐ `int64_t`
☐ `uint8_t`	☐ `uint16_t`	☐ `uint32_t`	☐ `uint64_t`

Riya wants to convert an array `A` from the previous task so that each element has type **`int`**. For example, if `A` contains the characters `{0x01 0xF1 0x80 0x02 0x36}`, the new array should contain `{0x00000001 0x000000F1 0x00000080 0x00000002 0x00000036}` — it's the same as `A` but each element is now an **`int`** instead of a **`char`**.

Riya's friend, Nikita, suggests that Riya converts her array by defining the following:

```
int *B = (int*)A;
```

Clearly indicate `B`, `B+1`, `B+2`, `B+3` and `B+4` in the following memory diagram.



1pt **4.3** Nikita's approach doesn't quite work. Help Riya out by writing a function that converts her array correctly. Fill in the missing types appropriately and make all casts explicit. As you do so, recall C's casting rules for numerical types:

- If the new type can represent the value, the value is preserved.
- If the new type can't represent the value,
  - if the new type is unsigned,
    - if the new type is smaller or the same, the least significant bits are retained.
    - if the new type is bigger, the bits are sign-extended.
  - if the new type is signed, the result is implementation-defined.

As earlier, if `A` contains the characters `{0x01 0xF1 0x80 0x02 0x36}`, the new array should contain `{0x00000001 0x000000F1 0x00000080 0x00000002 0x00000036}`.

```
int *char_array_to_ints(signed char *A, size_t num_elems) {




}
```

1pt **4.4** We want to convert an array of numbers of unspecified type (they could be **`char`**'s, `uint64_t`'s, ...) to an array of **`int`**'s. How to go about it? As long as we know the size of the array elements and how to turn this many bytes to an **`int`**, we can perform this conversion. Complete the code of the following function that does precisely this! You will want to use the fact that **`sizeof(char)`** `== 1`. Make all casts explicit.

```
typedef int elem2int_fn(void *e);

int *num_array_to_ints(void *A, elem2int_fn *F,
                       size_t elem_size, size_t num_elems) {




}
```