15-122: Principles of Imperative Computation, Spring 2020

Written Homework 12

Due on Gradescope: Thursday 23rd April, 2020 by 9pm

Name: ____

Andrew ID: ____

Section: ____

This written homework provides practice with C features such as pointer arithmetic, undefined behaviors and casting.

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.
- ***Preview***, the Mac's PDF viewer.
- ***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*.

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	Total
Points:	4	3	5	12
Score:				

1. **Pass by Reference Using C**

At various points in our C0 programming experience, we had to use somewhat awkward workarounds to deal with functions that need to return more than one value. Stack-allocated data structures and the address-of operator (&) in C give us a new way of dealing with this issue.

Sometimes, a function needs to be able to both 1) signal whether it can return a result, and 2) return that result if it is able to. Consider the following function parse_string that attempts to parse a string into an integer:

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

void parse_string(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);
  return;
}
```

The function parse_string relies on parse, a function which both sets *i to an integer equivalent to the integer pattern in *x (if possible) and also returns a boolean value of true if the parse succeeds, or false otherwise.

2pts

1.1 Using the address-of operator, rewrite the body of the parse_string function so that it does not heap-allocate, free, or leak any memory on the heap. You may assume parse has been implemented (its prototype is given above).

```
void parse_string(char *x) {
  REQUIRES(x != NULL);









  return;
}
```

2pts

1.2 In both C and C0, multiple values can be 'returned' by bundling them in a struct:

```
struct bundle {
  int part1;
  int part2;
};

struct bundle *split_int(int p) {
  struct bundle *A = xmalloc(sizeof(struct bundle));
  A->part1 = p>=0 ? 1 : -1;  // first value to be returned
  A->part2 = abs(p);         // second value to be returned
  return A;            // return both values together as a struct
}

int main() {
  ...
  struct bundle *B = split_int(-42);
  int sign  = B->part1;
  int value = B->part2;
  free(B);
  ...
}
```

Complete the declaration of the function split_int, as well as the snippet of main, to avoid heap-allocating, freeing, or leaking any memory on the heap. The rest of the code (...) should continue to behave exactly as it did before.

```
void split_int(____________________________________, int p) {
  A->part1 = p>=0 ? 1 : -1;
  A->part2 = abs(p);
  return;
}

int main() {
  ...
  struct bundle B;

  split_int(__________________________________________, -42);

  int sign  = ______________________________________________;

  int value = ______________________________________________;
  ...
}
```

2. C Program Behavior

Each of the following snippets of C code contains one or more errors. *Briefly* explain what is conceptually wrong with each example. No credit will be given if you simply copy error messages from the compiler, the runtime system, or `valgrind`. Of course you are encouraged to use these tools to help you understand the problems.

0.5pts

2.1

```
#include <stdio.h>
#include <string.h>
int main() {
  char *w;
  strcpy(w,"C programming");      // copy string into w
  printf("%s\n", w);
  return 0;
}
```

0.5pts

2.2

```
#include <stdlib.h>
#include "lib/xalloc.h"
#include "lib/contracts.h"

int main() {
  int* A = xmalloc(sizeof(int) * 10);
  for (int i = 1 ; i < 10 ; i++) {
    ASSERT(1 <= i);
    *(A + i) = i;
  }
  free(A+1);
  return 0;
}
```

0.5pts

2.3

```
#include <stdlib.h>
#include <stdio.h>
#include "lib/xalloc.h"
#include "lib/contracts.h"

int main() {
  int* A = xmalloc(sizeof(int) * 10);
  printf("Before: %d\n", A[0]);
  for (int i = 0 ; i < 10 ; i++) {
    ASSERT(0 <= i);
    A[i] = i;
  }
  printf("After: %d\n", A[0]);
  free(A);
  return 0;
}
```

0.5pts

2.4

```
#include <stdlib.h>
#include <stdio.h>
#include "lib/xalloc.h"
#include "lib/contracts.h"

int main() {
  int* A = xmalloc(sizeof(int) * 10);
  int* B = A+3;
  for (int i = 0 ; i < 10 ; i++) {
    ASSERT(0 <= i);
    A[i] = i;
  }
  free(A);
  printf("B: %d\n", *B);
  return 0;
}
```

0.5pts

2.5

```
#include "lib/contracts.h"

int oadd(int x, int y) {
  int result = x + y;
  if (x > 0 && y > 0) ASSERT(result > 0);
  if (x < 0 && y < 0) ASSERT(result < 0);
  return result;
}

// The omitted main function calls oadd
```

0.5pts

2.6

```
#include <stdio.h>
int main() {
  printf("DAVE: Open the pod bay doors please, HAL\n");
  char* hal = "I’m sorry Dave, I’m afraid I can’t do that.";
  printf("HAL: %s\n", hal);
  if (*hal = ’I’)
    printf("DAVE: Hello, HAL? Do you read me?\n");
  else
    printf("DAVE: What’s the problem?\n");
  return 0;
}
```

3. **Integer Types**

2pts

3.1 Suppose that we are working with 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).

We begin with the following declarations:

```
signed char the_char = -7;
unsigned char un_char_1 = 248;
unsigned char un_char_2 = 5;
int the_int = -247;
```

Fill in the table below. In the third column, always use two hex digits to represent a **char**, four hex digits to represent a **short**, and eight hex digits to represent an **int**. You might find these numbers useful: $2^8 = 256$, $2^{16} = 65536$ and $2^{32} = 4294967296$.

Most, but not all, of these answers can be derived from the lecture notes. If you can't find an answer from the lecture notes, you can look at online C references or just compile some code.

C expression	**Decimal value**	**Hexadecimal**
`the_char`	-7	0xF9
`(unsigned char) the_char`	249	0xF9
`(int) the_char`	-7	0xFFFFFFF9
`un_char_1`	248	
`(int)(signed char)un_char_1`		
`(int)(unsigned int)un_char_1`		
`un_char_2`	5	0x05
`(int)(signed char)un_char_2`		
`(int)(unsigned int)un_char_2`		
`the_int`	-247	
`(unsigned int)the_int`		
`(char)the_int`		
`(short)the_int`		
`(unsigned short)the_int`		

2pts

3.2 For this question, assume that **`char`** is a 1-byte signed integer type and that **`int`** is a 4-byte signed integer type.

Write the C function `condense` which takes a **`char`** array of length 4 and packs it into a single **`int`**. We want the 0th character aligned at the least significant byte, and the 3rd character aligned at the most significant byte. For example, given `F = {1, 2, -1, 4}`, `condense(F)` should return `0x04FF0201`.

For full credit,

- Make all casts explicit.
- Do not cast (or otherwise convert types) directly between signed and unsigned types of different sizes.
- Do not rely on the *endianness*[1] of your machine. For example, the following code is incorrect:

  ```
  int condense(char* F) { return *((int*) F); }
  ```

- Make sure your solution works for **`char`** arrays containing negative values.
- Write code which is clear and straightforward.

```
int condense(char *F) {











}
```

[1] "Endianness" refers to the natural storage order of bytes for a particular hardware architecture; you can read about it on Wikipedia, and don't forget to read *Gulliver's Travels* in your no doubt copious spare time.

1pt

3.3 Suppose we've defined the following functions:

```
int fib(int n);  // returns the nth fibonacci number
int cat(int n);  // returns the nth catalan number
int las(int n);  // returns the nth look-and-say number
```

Complete the code below such that it will print

```
2 3 5 0 1 1
5 14 42 1 1 2
1211 111221 312211 1 11 21
```

(*Hint:* The **typedef** on the first line should define the type int2int_fn. This type should match the type of a function such as fib, cat, or las.)

```
typedef ______________________________________________________;

void map_print(int2int_fn* f, int* A, size_t n) {
  for (size_t i = 0; i < n; i++) {

    int x = __________________________________________________;
    printf("%d ", x);
  }
  printf("\n");
}

int main() {
  int A[6] = {3, 4, 5, 0, 1, 2};

  map_print(____________________________________________, A, 6);

  map_print(____________________________________________, A, 6);

  map_print(____________________________________________, A, 6);
  return 0;
}
```