15-122 : Principles of Imperative Computation

Fall 2012

Assignment 7 – Selected Solutions

(Theory Part)

The following solutions are provided to you to help you study this semester. They are not to be distributed to others outside of the class nor are they intended to be used by students in future semesters as a substitute for completing one's own homework assignments.

1. **C programming issues**

   For each of the following problems, state what is wrong with the code and show how to correct it. Do not just try to compile it and write down the error message. (Some of these will compile without error, and some will even run and produce output, but they all contain conceptual errors that may affect correctness.) Read the code and explain what is being done wrong, conceptually. Think about all the ways to incur undefined behavior in C, including accessing unallocated or uninitialized memory, dereferencing `NULL`, dividing by zero, and arithmetic overflow.

(1) (a)
```
#include <stdio.h>
#include <string.h>

int main() {
  char *w;
  strcpy(w,"C programming");
  printf("%s\n", w);
  return 0;
}
```

Solution: `w` is not initialized.

It is possible to fix this by adding a call to `malloc` to initialize `w` and then freeing that memory later, but this is more expensive and more error-prone than simply using stack allocated memory as in the sample solution below.

```
int main() {
  // notice extra character for null terminator
  // (often, you'll see much larger than needed buffers in C code)
  char w[14];
  strcpy(w, "C programming");
  printf("%s\n", w);
  return 0;
```

```
}
```

(1) (b)

```
#include "xalloc.h"

int main() {
  int* a = xmalloc(100);
  for (int i=0; i<100; i++)
    a[i] = i;
  return 0;
}
```

Solution: We have not allocated enough memory. This gives us only 100 bytes and each integer requires more than one byte.

```
#include "xalloc.h"
[
int main() {
  int* a = xmalloc(100 * sizeof(int));
  for (int i=0; i<100; i++)
    a[i] = i;
  return 0;
}
```

(1) (c)

```
#include "xalloc.h"
#include <string.h>

int main() {
  char* name = xmalloc(strlen("wordpress"+1));
  strcpy(name, "wordpress");
  return 0;
}
```

Solution: Unfortunately, `"wordpress"+1` is not a syntax error. `"wordpress"` is a `char*` and we can increment that pointer by 1 so that it points to the next character. This means we're actually allocating enough memory for `"ordpress"`. Therefore, `strcpy(name, "wordpress")` will trigger undefined behavior in the form of a buffer overflow – we are *two* characters short from what we need in the `name` buffer.

Additionally, there's a memory leak, but we didn't require that you identify or fix that bug to get full points.

```
#include "xalloc.h"
#include <string.h>

int main() {
  char* name = xmalloc(strlen("wordpress")+1);
  strcpy(name, "wordpress");
  free(name);
  return 0;
}
```

(1) (d) The standard string library function `strncpy(dest, src, n)` copies the specified number of characters `n` from the source string `src` to the destination string `dest`.

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

int main() {
  char *letter_data = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  char a[16];
  strncpy(a, letter_data, sizeof(a));
  printf("The first sixteen letters are: %s\n", a);
  return 0;
}
```

Solution: The most important bug in this code that we wanted you to notice was that we want `a` to hold 16 letters, but the buffer is only 16 characters long, meaning we didn't leave any room for the null terminator required by C strings. If you noticed this and corrected 16 to 17, that was sufficient to get full points – but there's actually far more going on here.

Next, `strncpy` does *not* add a null character to the end of the destination string even if there is enough room (you can read `man strncpy`). It will only copy over a null character if it encounters it as the last character of the source string. This means we need to manually null terminate `a`.

The use of `sizeof` on a stack allocated array is actually fine and will correctly return 16 in the original code. This is because the size of the array `a` is known at compile time (unlike an array created with `malloc`, which could be of arbitrary size). If you find this surprising, you may want to re-read the documentation for `sizeof` – it has many special-case behaviors including statically allocated arrays versus runtime allocated arrays, allocation of structs, and allocation of structs that include an array of unknown size at the end of the struct (do you remember what `sizeof` will return for such a struct?).

There are no possible memory leaks here since all arrays are stack allocated or otherwise local variables.

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

int main() {
  char *letter_data = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  char a[17];
  strncpy(a, letter_data, sizeof(a)-1);
  a[16] = '\0';
  printf("The first sixteen letters are: %s\n", a);
  return 0;
}
```

(1) (e) This code fragment shows a C function that is called from another function. It is supposed to return the result only if no overflow occurs.

```
#include <assert.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;
}
```

Solution: Signed integer overflow is undefined in C. We gave you full points simply for identifying this issue. To correctly fix this issue, you must check if overflow *will* occur *before* it occurs:

```
#include <assert.h>
#include <limits.h>

int oadd(int x, int y) {
   REQUIRE(    (x >  0 && y >  0 && x <= INT_MAX - y)
            || (x <  0 && y <  0 && x >= INT_MIN - y)
            || (x >= 0 && y <= 0)
            || (x <= 0 && y >= 0);
   return x + y;
}
```

(1) (f) This code fragment shows a C function that is called from another function.

```
#define TABLESIZE 100
int table[TABLESIZE];

int insert_in_table(int pos, int value) {
  if (pos >= TABLESIZE)
    return -1;
  table[pos] = value;
  return 0;
}
```

Solution: We need to check if `pos` is negative.

Global variables such as `table` are perfectly legal in C.

```
#define TABLESIZE 100
int table[TABLESIZE];

int insert_in_table(int pos, int value) {
  if (pos >= TABLESIZE || pos < 0)
    return -1;
  table[pos] = value;
  return 0;
}
```

(1) (g) This code fragment shows a C function that is used inside the library implementation of stacks. Assume that `is_stack` returns true.

```
#include "contracts.h"
#include "xalloc.h"

int stack_size(stack S) {
  REQUIRES(is_stack(S));
  list *L = xmalloc(sizeof(struct list_node));
  int size = 0;
  for (L = S->top; L != S->bottom; L = L->next)
    size++;
  return size;
}
```

Solution: `L` is allocated and the address is the immediately overwritten by `S->top`, meaning that we forever lose the ability to free the allocated memory, which guarantees a leak. Calling `free` at the end of the function actually makes the situation worse by freeing a pointer that `stack_size` doesn't own (remember the concept of pointer ownership).

The best fix in this situation is to never needlessly allocate this memory in the first place:

```
#include "contracts.h"
#include "xalloc.h"

int stack_size(stack S) {
  REQUIRES(is_stack(S));
  int size = 0;
  for (list* L = S->top; L != S->bottom; L = L->next)
    size++;
  return size;
}
```

(1) (h) This code fragment shows a C function that applies a function to an array of length `n`. We would expect it to be called from another function.

```
#include <stdlib.h>

int* apply_fn(int (*f)(int* x, int y), int n, int A[n]) {
  int accum = 0;
  int* result = NULL;
  for (int i = 0; i < n; i++) {
    accum = (*f)(&accum, A[i]);
  }
  result = &accum;
  return result;
}
```

Solution: The biggest issue with this function is that it's returning (indirectly) a pointer to `accum`, which is a stack allocated variable. Any use of this variable outside of `apply_fn` is undefined since `accum` will be popped off the stack frame when `apply_fn` returns. We should really just return `accum` by value instead.

We didn't check if the function pointer `f` is null. If you noticed and fixed this, we also gave you credit.

Passing `int A[n]` is actually valid syntax in C. It has essentially the same semantics as `int* A` since the size of the array `A` can't be checked at compile time. Despite the fact that C allows you to use this syntax, it won't actually check that the length of the array `A` has length `n`.

```
#include <stdlib.h>

int apply_fn(int (*f)(int* x, int y), int n, int A[n]) {
  REQUIRE(f != NULL);
  int accum = 0;
  for (int i = 0; i < n; i++) {
    accum = (*f)(&accum, A[i]);
  }
  return accum;
}
```