Lecture 18
C's Memory Model

15-122: Principles of Imperative Computation (Summer 2024)
Frank Pfenning, Rob Simmons

May 27, 2024

In this lecture, we'll talk about the differences between the memory model we developed in C0, with local and allocated sections, and the memory model that C uses. Much of the model will stay the same, but we'll go more in depth and learn some new terms.

Additional Resources

- Review slides (`https://cs.cmu.edu/~15122/handouts/slides/review/20-cmem.pdf`)
- Code for this lecture (`https://cs.cmu.edu/~15122/handouts/code/20-cmem.tgz`)

1 The C0 and C Memory Model

When we talk about memory in C0, C1, and C, that memory is always in one of three places:

- *Local variables* (including the arguments to functions) are stored in memory. In both C0 and C, this memory is reserved automatically when we declare a new local variable, though in C the contents of that local memory aren't initialized. The local memory gets reclaimed as soon as the local variable goes out of scope.

- *Allocated memory* was reserved with **`alloc`** and **`alloc_array`**. We always accessed this memory by referring to its address (the address stored in a non-`NULL` pointer or the address of an array). When we reserved allocated memory in C0, this memory was initialized to default values for us. In C, `xmalloc` does not initialize memory.

- In C1, we said that the compiled code for functions was stored in *read-only memory*, and we could get pointers into that read-only memory by taking the address of a function: `&f`. Many more things in C are stored in read-only memory, like string literals.

In this class, we think about *all* of this data residing in memory.[1] In this picture, memory is comprised of an enormous array of bytes, where each index in the array of bytes is an address. The lowest address is 0, and the highest address on most modern, 64-bit processors is $2^{64} - 1 =$ `0xFFFFFFFFFFFFFFFF`. This is an almost inconceivably large array of bytes, far larger than the physical RAM installed in *any* computer, but the operating system plays tricks to allow processors to pretend like they have access to this enormous array. One way this is done is by only giving programs access to individual *segments* of this array. Modern hardware prevents individual running programs from accessing memory outside their allocated segments. (This is a good thing: it means that, no matter how much you mess up while programming in C, it's going to be difficult for you to interfere with other running programs like your text editor or virus scanner.)

The memory used for the local variables of a function is allocated and de-allocated according to a stack discipline, so we call this portion of memory the *stack*. Memory that we explicitly allocate is reserved in what we call the *heap*, though it has no relationship to the heap data structure. Read-only memory is divided into the *TEXT* region which contains the compiled code of the running program and the *DATA* region which contains string literals. Therefore, the big picture of memory looks like this:

```
0xFFFFFFFFFFFFFFFF        |
                          | OS AREA
                          | ============
                          | System stack   (local variables)
                          | ============
                          | unused
                          | ============
                          | System heap    (allocated memory)
                          | ============
                          | TEXT and DATA  (read only memory)
                          | ============
                          | OS AREA
0x0000000000000000        |
```

[1]Later classes will complicate this picture by talking about things like *registers* that you don't need to know about in 15-122.

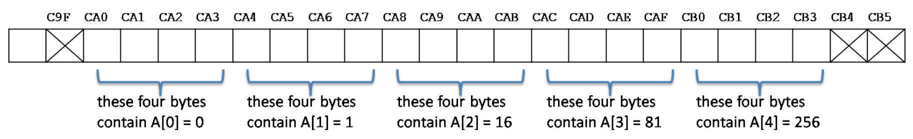
One consequence of this memory layout is that the stack grows towards the heap, and the heap usually grows towards the stack. Programs cannot access addresses (indices in this enormous array) that belong to the operating system. If they try, programs get an "exception" like a segmentation fault. C0 takes great care to ensure that it never gives you any pointers to uninitialized or random or garbage data in memory, *except*, of course, the NULL pointer. NULL is a special pointer to the memory address 0, which belongs to the operating system. Attempting to read from or write to a NULL pointer is undefined behavior, which we'll discuss more in a later section. In practice, trying to access such a pointer will cause us to access a segment of memory we don't have permissions to — this results in a *segmentation fault*, or segfault.

2 Arrays and Pointer Arithmetic

When compared to C0, the most shocking difference is that C does not distinguish arrays from pointers. We allocate enough space for a single integer by writing **sizeof**(**int**), and we allocate enough space for an array of 5 integers by just multiplying the size of a single integer by 5.

```
int *A = xmalloc(sizeof(int) * 5);
for (size_t i = 0; i < 5; i++) A[i] = i*i*i*i;
```

Assuming 4-byte integers, this 5 element array is treated by C as no more and no less than a single 20 byte memory segment that we are allowed to use. If the call to malloc returned the memory address 0xCA0, then after the **for** loop is done, the four bytes of memory addressed by 0xCA8 to 0xCAB will together represent the integer 16, the contents of the third index of the array, A[2], and so on:


C9F
CA0
CA1
CA2
CA3
CA4
CA5
CA6
CA7
CA8
CA9
CAA
CAB
CAC
CAD
CAE
CAF
CB0
CB1
CB2
CB3
CB4
CB5
these four bytes contain A[0] = 0
these four bytes contain A[1] = 1
these four bytes contain A[2] = 16
these four bytes contain A[3] = 81
these four bytes contain A[4] = 256

One consequence of this conflation of pointers and arrays is that writing *A and writing A[0] necessarily means the same thing. We can also add an integer to a pointer, but this modifies the pointer not in terms of bytes but in terms of array elements. This allows us to get pointers into arrays!

```
int *x = A + 3;
*x += 5;
```

In the example above, then after running these two lines, the local variable x will contain the pointer 0xCAC, and the assignment will cause both *x and A[3] to evaluate to 86 instead of 81. This is a form of aliasing that was impossible in C0, but it is relatively common in C. But because only the pointer 0xCA0 in the example above was returned from xmalloc, only that pointer should be freed: it would be a memory error to free the pointer 0xCAC stored in x.

When we allocate very large arrays, we may want them allocated with default values, the way we did in C0. The C standard library provides a function, calloc, to do just that, and our xalloc.h library has a non-NULL-returning xcalloc version of calloc that you should use. With xcalloc, we can allocate an array of seven elements, all of which are initialized to 0, like this:

```
int *B = xcalloc(7, sizeof(int));
for (size_t i = 1; i < 7; i++) A[i] = A[i-1]*2 + 3;
```

The only differences between xmalloc and xcalloc is that the latter initializes the memory to be all zeroes and takes two arguments. The xcalloc function takes two sizes n and m and allocates $n \times m$ bytes. We think of xmalloc as being like C0's **alloc** and we think of xcalloc as being like C0's **alloc_array**, but in C you can allocate arrays and pointers with either xmalloc or xcalloc.

3 Undefined Behavior

We have described the following as memory errors:

- Reading uninitialized memory (on the stack or on the heap).
- Dereferencing memory outside of a valid allocated segment (which includes NULL pointer dereference, array out of bounds errors), or trying to read or write to an **int** when you've only allocated enough size for a **char**.
- Writing to memory that is in a read-only segment like TEXT and DATA.
- Using a memory allocation that has been freed, double-freeing a pointer, or freeing any pointer that wasn't returned from xmalloc or xcalloc.

In C0, memory errors would always predictably and consistently cause the program to stop executing. In C, this is *definitely not the case*.

Array accesses are not checked at all, and out-of-bounds memory references (whose result is formally undefined) may lead to unpredictable results. The program might stop with an error, or keep going, but after undefined behavior occurs it is difficult, if not impossible, to predict a program's behavior. For example, the code fragment

```
int main() {
  int* A = xmalloc(sizeof(int));
  A[0] = 0;          /* ok - A[0] is like *A */
  A[1] = 1;          /* error - not allocated */
  A[317] = 29;       /* error - not allocated */
  A[-1] = 32;        /* error - not allocated(!) */
  printf("A[-1] = %d\n", A[-1]);
  return 0;
}
```

will not raise any compile time error or even warnings, even under the strictest settings. Here, the call to `xmalloc` allocates enough space for a single integer in memory. In this class, we are using `gcc` with all our standard flags:

```
% gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g
```

The code above executes ok, and in fact prints `32`, despite four blatant errors in the code.

To discover whether such errors may have occurred at runtime, we can use the `valgrind` tool.

```
% valgrind ./a.out
...
==nnnn== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
```

which produces useful error messages (elided above) and indeed, flags errors in code that didn't appear to have any errors when we ran it without `valgrind`.

You can also guard memory accesses with appropriate **`assert`** statements that abort the program when attempting out-of-bounds accesses.

There's an old joke that whenever you encounter undefined behavior, your computer could decide to play *Happy Birthday* or it could catch on fire. This is less of a joke considering recent events:

- In 2010, Alex Halderman's team at the University of Michigan successfully hacked into Washington D.C.'s prototype online voting system, and caused its web page to play the University of Michigan fight song, "The Victors."[2]

[2]Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. *Attacking the Washington, D.C. Internet Voting System*. Proceedings of the 16th Conference on Financial Cryptography and Data Security, February 2012.

- The Stuxnet worm caused centrifuges, such as those used for uranium enrichment in Iran, to malfunction, physically damaging the devices.[3]

Not quite playing *Happy Birthday* and catching on fire, but close enough.

4 Address-of

In our C0 and C memory model, almost *everything* has an address. If `e` is an expression (like `x`, `A[12]`, `*x`, `A.fld`, or `A->fld`) that describes a memory location which we can read from and potentially write to, then that memory location exists in memory as some number of bytes with an address. Writing `&e` then gives us a *pointer* to that memory location. In C0, if we have a struct containing a string and an integer, it's not possible to get a pointer to *just* the integer. This is possible in C:

```
struct wcount {
  char *word;
  int count;
};

void increment(int *p) {
  REQUIRES(p != NULL);
  *p = *p + 1;
}

void increment_count(struct wcount *wc) {
  REQUIRES(wc != NULL);
  increment(&(wc->count));
}
```

Because the type of `wc->count` is **`int`**, the expression `&(wc->count)` is a pointer to an **`int`**. Calling `increment_count(B)` on a non-null struct will cause the `count` field of the struct to be incremented by the `increment` function, which is passed a pointer to the second field of the struct.

Because of the address-of operation, we never have to actually use pointer arithmetic if we don't want to. For example, `&A[3]` is equivalent to `A + 3`, and much more readable.

[3]Holger Stark. *Stuxnet Virus Opens New Era of Cyber War*. Spiegel Online, August 8, 2011.

5 Stack Allocation

In C, we can also allocate data on the *system stack* (which is different from the explicit stack data structure you have studied). Each function allocates memory in its so-called *stack frame* for local variables. We can obtain a pointer to this memory using the address-of operator. For example:

```
int main() {
  int a1 = 1;
  int a2 = 2;
  increment(&a1);
  increment(&a2);
  ...
}
```

Note that there is no call to `xmalloc` or `xcalloc` which allocate spaces on the system heap (again, this is different from the heap data structure we used for priority queues).

We can only free memory allocated with `xmalloc` or `xcalloc`, but not memory that is on the system stack. Such memory will automatically be freed when the function whose frame it belongs to returns. This has two important consequences. The first is that the following is a bug, because `free` will try to free the memory holding `a1`, which is not on the heap:

```
int main() {
  int a1 = 1;
  int a2 = 2;
  free(&a1);  // BUG: trying to free a local variable
  ...
}
```

The second consequence is pointers to data stored on the system stack do not survive the function's return. For example, the following is a bug:

```
int *f_ohno() {
  int a = 1;  // BUG: a is deallocated when f_ohno() returns
  return &a;
}
```

(recent versions of `gcc` recognize that this makes little sense and issue a compilation error). A correct implementation requires us to allocate on the system heap, using a call to `malloc` or `calloc` (or one of the library functions which calls them in turn).

```
int *f() {
  int* x = xmalloc(sizeof(int));
  *x = 1;
  return x;
}
```

C offers a special syntax to allocate *arrays* on the stack. The following `main` declares an array `A` with 5 elements, initializes it, calls the function `sum` which adds up its elements, and prints the result.

```
int sum(int* A, int n) {
  int res = 0;
  for (int i = 0; i < n; i++)
    res += A[i];
  return res;
}

void main() {
  int A[5];
  for (int i=0; i < 5; i++)
    A[i] = i;
  printf("%d\n", sum(A, 5));
}
```

C even provides syntax to initialize a literal array, i.e., an array whose length and initial contents are fixed values. The above `main` function can indeed be rewritten as follows:

```
void main() {
  int A[] = {0, 1, 2, 3, 4};
  printf("%d\n", sum(A, 5));
}
```

Note that it's not even necessary to specify the length of `A`: the compiler figures it out.

When stack allocation is possible, it can be a real benefit, because it saves you from having to remember to free memory explicitly. However, if the data structure we allocate needs to survive past the end of the current function you *must* allocate it on the heap.

6 Exercises

Exercise 1 (sample solution on page 18). *Translate the following C0 implementation of UBA's for strings into C. Make sure to think about whether some functions are still necessary, and whether you need additional functions. Make a separate header file for the interface. You may assume that the functions seen in class for C0-style contracts and safe allocation are declared in* `lib/contracts.h` *and* `lib/xalloc.h`*, respectively.*

```
#use <util>
#use <string>
#use <conio>

/**************************************************************************/
/************************* BEGIN IMPLEMENTATION *************************/

typedef struct uba_header uba;
struct uba_header {
  int size;          // 0 <= size && size < limit
  int limit;         // 0 < limit
  string[] data;     // \length(data) == limit
};

// Internal debugging function that prints an UBA without checking contracts
// (useful to debug representation invariant issues)
void uba_print_unsafe(uba* A)
{
  printf("UBA limit=%d; length=%d; data=[", A->limit, A->size);
  for (int i = 0; i < A->limit; i++)
  //@loop_invariant 0 <= i && i <= A->limit;
  {
    if (i < A->size)
      printf("%s", A->data[i]);
    else
      printf("X");
    if (i < A->limit-1) {
      if (i == A->size-1) printf(" | ");
      else printf(", ");
    }
  }
  printf("]");
}

bool is_array_expected_length(string[] A, int length) {
  //@assert \length(A) == length;
  return true;
}
```

```
bool is_uba(uba* A) {
  return A != NULL
      && 0 <= A->size && A->size < A->limit
      && is_array_expected_length(A->data, A->limit);
}

// Internal debugging function that prints an SSA
// (useful to spot bugs that do not invalidate the representation invariant)
void uba_print_internal(uba* A)
//@requires is_uba(A);
{
  uba_print_unsafe(A);
}


// Implementation of exported functions
int uba_len(uba* A)
//@requires is_uba(A);
//@ensures 0 <= \result && \result < \length(A->data);
{
  return A->size;
}

string uba_get(uba* A, int i)
//@requires is_uba(A);
//@requires 0 <= i && i < uba_len(A);
{
  return A->data[i];
}

void uba_set(uba* A, int i, string x)
//@requires is_uba(A);
//@requires 0 <= i && i < uba_len(A);
//@ensures is_uba(A);
{
  A->data[i] = x;
}

uba* uba_new(int size)
//@requires 0 <= size;
//@ensures is_uba(\result);
//@ensures uba_len(\result) == size;
{
  uba* A = alloc(uba);
  int limit = size == 0 ? 1 : size*2;
  A->data = alloc_array(string, limit);
  A->size = size;
  A->limit = limit;

  return A;
}

void uba_resize(uba* A, int new_limit)
/* A may not be a valid array since A->size == A->limit is possible! */
//@requires A != NULL;
//@requires 0 <= A->size && A->size < new_limit;
//@requires \length(A->data) == A->limit;
//@ensures is_uba(A);
{
  string[] B = alloc_array(string, new_limit);

  for (int i = 0; i < A->size; i++)
  //@loop_invariant 0 <= i;
  {
    B[i] = A->data[i];
  }

  A->limit = new_limit;
  A->data = B;
}

void uba_add(uba* A, string x)
//@requires is_uba(A);
//@ensures is_uba(A);
{
  A->data[A->size] = x;
  (A->size)++;

  if (A->size < A->limit) return;
  assert(A->limit <= int_max() / 2); // Fail if array would get too big
  uba_resize(A, A->limit * 2);
}

string uba_rem(uba* A)
//@requires is_uba(A);
//@requires 0 < uba_len(A);
//@ensures is_uba(A);
{
  (A->size)--;
  string x = A->data[A->size];

  if (A->limit >= 4 && A->size <= A->limit / 4)
    uba_resize(A, A->limit / 2);
  return x;
}

void uba_print(uba* A)
//@requires is_uba(A);
{
  printf("[");
  for (int i = 0; i < A->size; i++)
    {
      printf("%s", A->data[i]);
      if (i+1 != A->size) printf(", ");
    }
  printf("]");
}


// Client type
typedef uba* uba_t;

/************************** END IMPLEMENTATION *************************/
/*********************************************************************/

/*********************************************************************/
/******************************* Interface *****************************/

// typedef ______* uba_t;

int uba_len(uba_t A)
/*@requires A != NULL;               @*/
/*@ensures \result >= 0;             @*/ ;

uba_t uba_new(int size)
/*@requires 0 <= size;               @*/
/*@ensures \result != NULL;          @*/
/*@ensures uba_len(\result) == size; @*/ ;

string uba_get(uba_t A, int i)
/*@requires A != NULL;                @*/
/*@requires 0 <= i && i < uba_len(A); @*/ ;

void uba_set(uba_t A, int i, string x)
/*@requires A != NULL;                @*/
/*@requires 0 <= i && i < uba_len(A); @*/ ;

void uba_add(uba_t A, string x)
/*@requires A != NULL;                @*/ ;
```

```
string uba_rem(uba_t A)
/*@requires A != NULL;                @*/
/*@requires 0 < uba_len(A);           @*/ ;

// bonus function
void uba_print(uba_t A)
/*@requires A != NULL;                @*/ ;
```

Exercise 2 (sample solution on page 24). *Identify all lines in the following code with undefined behavior, and correct those issues. Try to do this without using Valgrind.*

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

typedef struct rectangle rect;
struct rectangle {
  int x1;
  int y1;
  int x2;
  int y2;
};

int calc_area(rect* r) {
  int* area;
  *area = (r->x2 - r->x1) * (r->y2 - r->y1);
  free(r);
  return *area;
}

int main() {
  rect r1;
  r1.x1 = 0;
  r1.y1 = 0;
  r1.x2 = 4;
  r1.y2 = 5;
  int area1 = calc_area(&r1);

  rect* r2 = xmalloc(sizeof(rect));
  int area2 = calc_area(r2);
  if (area2 > area1) return area2;
  return area1;
}
```

Exercise 3 (sample solution on page 25). *Use Valgrind to find and correct the issues in the following code (the* `--leak-check=full` *and* `track-origins=yes` *flags will be very useful for this).*

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

struct point {
  int x;
  int y;
};

int main () {
  int len = 63;
  struct point **arr = xmalloc(sizeof(struct point*) * len);
  for (int i = 0; i < len/2; i++) {
    arr[2*i] = xmalloc(sizeof(struct point));
    arr[2*i + 1] = xmalloc(sizeof(struct point));
    arr[2*i]->x = i+1;
    arr[2*i]->y = i+1;
    arr[2*i+1]->x = i/2;
    arr[2*i+1]->y = 5*i+3;
  }

  for (int i = 0; i < len; i++) {
    if (arr[i]->x + 100 < arr[i+1]->y) {
      printf("%s\n", "Small x value");
    }
    free(arr[i]);
  }
}
```

Exercise 4 (sample solution on 27). *The function* `count_words` *counts the number of words in a file. Upon successful return, the function returns 0, indicating no error, and the number of words is stored at the address referenced by* `num_words`*. If there is an error reading the file, a nonzero error value is returned.*

```
error_t count_words(file_t f, size_t *num_words);
//@requires num_words != NULL;
```

Complete the main function below such that the local variable `n` *will contain the number of words in the file* `new_text` *without using heap allocated memory.*

```
int main() {
  // ...
  // assume file_t new_text has been previous initialized

  ________________;

  error_t e = count_words(_________, __________);
  if (e != 0) printf("count_words failed with error %d.\n", e);
  else printf("The text file contains %zu words", n);

  // ...
}
```

Exercise 5 (sample solution on 28). *Rewrite this code so that it does not use pointer arithmetic.*

```
int main () {
  int  *A = xcalloc(sizeof(int),  100);
  int **B = xcalloc(sizeof(int*), 100);
  int  *C = xcalloc(sizeof(int),  100);

  for (int i = 0; i < 100; i++) {
    *(A + i) = i*i;
    *(B + i) = A + i;
    *(C + i) = **(B + i);
  }
  return 0;
}
```

Exercise 6 (sample solution on 28). *Pointer arithmetic is error-prone and should be avoided. It is instructive to know about it as you may encounter it in code written by others. Replace all uses of the bracket notation to access an element of an array with pointer arithmetic (e.g.,* `A[0]` *with* `*a`*).*

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

void mult2(int* x) {
  *x = *x * 2;
}

int main(){
  int* arr = xmalloc(sizeof(int)*5);
  for (int i = 0; i < 5; i++) {
    arr[i] = i;
  }
  for (int j = 0; j < 5; j++) {
    mult2(&arr[j]);
  }

  free(arr);
  return 0;
}
```

Exercise 7 (sample solution on 29). *Consider the following code*

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

void set_values(int *arr, size_t n) {
  for (size_t i = 0; i < n; i++) {
    arr[i] = 2*i;
  }
}

int main () {
  int *A = xmalloc(sizeof(int)*15);
  set_values(A, 15);
  for (size_t i = 0; i < 15; i++) {
    printf("%d\n", A[i]);
  }
  free(A);
  return 0;
}
```

Does this code behave differently if we instead create `A` as `int A[15]`? If so, what about the code is different, and what needs to be changed to make it work the same as before?

Exercise 8 (sample solution on 29). *One of the consequences of having stack allocation in C is that we can assign a struct to another of the same type. Doing so copies the fields of the former to the fields in the latter. This can come really handy.*

We are given an unknown struct type, `struct foobar`. We do not know any of the fields of this struct, but we are asked to write a function that takes in two *variables of type* `struct foobar*` *and swaps the values that the two point to. How can we do this, and was this possible in C0? If not, why can we do this in C?*

Exercise 9 (sample solution on 30)**.** *Implement the function* `str_shorten` *that takes a string and a length, and truncates the string based on the given new length. You may assume that the given length is less than or equal to the original length of the string. Your function should have the following type:*

```
char *str_shorten(char *s, size_t len)
/*@requires len <= strlen(s); @*/ ;
```

Exercise 10 (sample solution on 30)**.** *Write the function interweave, that takes two strings and interweaves their characters (e.g.,* `"ab"` *and* `"cd"` *becomes* `"acbd"`*). The returned string should be in allocated memory and the inputs should not be modified. Your function should have the following type:*

```
char *interweave(char *s1, char *s2);
```

Sample Solutions

Solution of exercise 1

First, we need to create the header file with the interface. As usual, we need a header guard (using the macro definition of `UBA_H`). We also have to provide a **`typedef`** for `uba_t`, in order for the type to be defined when we use it later in the file. While this does tell the client that we are using a struct for our implementation, it does not reveal any of the fields. Finally, notice that we need to add a `uba_free` function to the interface, since `uba_new` allocates memory to return, so we need to provide a function for the client to free the UBA when they are done with it. The resulting header file is

```
#ifndef UBA_H
#define UBA_H

typedef struct uba_header* uba_t;

size_t uba_len(uba_t A)
/*@requires A != NULL;                @*/ ;

uba_t uba_new(size_t size)
/*@ensures \result != NULL;           @*/
/*@ensures uba_len(\result) == size;  @*/ ;

char *uba_get(uba_t A, size_t i)
/*@requires A != NULL;                @*/
/*@requires i < uba_len(A);           @*/ ;

void uba_set(uba_t A, size_t i, char* x)
/*@requires A != NULL;                @*/
/*@requires i < uba_len(A);           @*/ ;

void uba_add(uba_t A, char *x)
/*@requires A != NULL;                @*/ ;

char *uba_rem(uba_t A)
/*@requires A != NULL;                @*/
/*@requires 0 < uba_len(A);           @*/ ;

void uba_print(uba_t A)
/*@requires A != NULL;                @*/ ;

void uba_free(uba_t A)
/*@requires A != NULL;                @*/ ;

#endif
```

The actual `uba.c` file is very similar to the original C0 file, with a few small differences that we will now highlight. Let's start with the first few lines of this file.

This file starts by including some of the standard C libraries (used by the rest of the translation), our usual custom libraries, and the header files for UBA's.

We set type of any integer used to index arrays to `size_t`. We do the same for bounds on such indices, here the fields `size` and `limit`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#include <stdint.h>
#include <assert.h>
#include "lib/contracts.h"
#include "lib/xalloc.h"
#include "uba.h"

typedef struct uba_header uba;
struct uba_header {
  size_t size;          // size < limit
  size_t limit;         // 0 < limit
  char **data;          // \length(data) == limit
};
```

Next we translate the representation invariant function `is_uba` and the other helper functions. Since is not possible to check the length of an array in C, we get rid of the `is_array_expected_length` function.

Note that we don't have to change any of the printing, since the original code used `printf` which we have in C, but if different print functions were used we would need to change them to `printf`.

```
// Internal debugging function that prints an UBA without checking contracts
// (useful to debug representation invariant issues)
void uba_print_unsafe(uba *A) {
  printf("UBA limit=%zu; length=%zu; data=[", A->limit, A->size);
  for (size_t i = 0; i < A->limit; i++)
  {
    if (i < A->size)
      printf("%s", A->data[i]);
    else
      printf("X");
    if (i < A->limit-1) {
      if (i == A->size-1) printf(" | ");
      else printf(", ");
    }
  }
  printf("]");
}

bool is_uba(uba *A) {
  return A != NULL && A->size < A->limit;
}

// Internal debugging function that prints an SSA
// (useful to spot bugs that do not invalidate the representation invariant)
void uba_print_internal(uba *A) {
  REQUIRES(is_uba(A));
  uba_print_unsafe(A);
}
```

Next are the basic functions exported by the interface. When allocating arrays and structs, we have to use one of `xmalloc/xcalloc`. We translate the contracts directly.

```
// Implementation of exported functions
size_t uba_len(uba *A) {
  REQUIRES(is_uba(A));
  return A->size;
}

uba *uba_new(size_t size) {
  uba *A = malloc(sizeof(uba));
  size_t limit = size == 0 ? 1 : size*2;
  A->data = xmalloc(sizeof(char*) * limit);
  A->size = size;
  A->limit = limit;

  ENSURES(is_uba(A));
  ENSURES(uba_len(A) == size);
  return A;
}

char *uba_get(uba *A, size_t i) {
  REQUIRES(is_uba(A));
  REQUIRES(i < uba_len(A));
  return A->data[i];
}

void uba_set(uba *A, size_t i, char *x) {
  REQUIRES(is_uba(A));
  REQUIRES(i < uba_len(A));
  ENSURES(is_uba(A));
  A->data[i] = x;
}

void uba_print(uba* A) {
  REQUIRES(is_uba(A));
  printf("[");
  for (size_t i = 0; i < A->size; i++) {
    printf("%s", A->data[i]);
    if (i+1 != A->size) printf(", ");
  }
  printf("]");
}
```

In `uba_resize` when we are done with the old array, we need to free it before we switch to the new array, to avoid losing access to the old array and leaking it. We omit the translation of the loop invariants, although it can easily be emulated with ASSERTs.

```
void uba_resize(uba *A, size_t new_limit) {
/* A may not be a valid array since A->size == A->limit is possible! */
  REQUIRES(A != NULL);
  REQUIRES(A->size < new_limit);
  char **B = (char **)xmalloc(sizeof(char*)* new_limit);

  for (size_t i = 0; i < A->size; i++)
    B[i] = A->data[i];
  A->limit = new_limit;
  free(A->data);
  A->data = B;
  ENSURES(is_uba(A));
}

void uba_add(uba *A, char *x) {
  REQUIRES(is_uba(A));
  A->data[A->size] = x;
  (A->size)++;

  if (A->size < A->limit) return;
  assert(A->limit <= SIZE_MAX / 2); // Fail if array would get too big
  uba_resize(A, A->limit * 2);
  ENSURES(is_uba(A));
}

char *uba_rem(uba *A) {
  REQUIRES(is_uba(A));
  REQUIRES(0 < uba_len(A));
  (A->size)--;
  char *x = A->data[A->size];

  if (A->limit >= 4 && A->size <= A->limit / 4)
    uba_resize(A, A->limit / 2);
  ENSURES(is_uba(A));
  return x;
}
```

The new function `uba_free` frees a UBA to avoid memory leaks. Given a UBA `A`, we will have it free two pieces of data:

1. the array `A->data`, and
2. the struct `A` itself.

`uba_free` does not free the strings contained in the array — it will be the client's responsibility to do so.

Note that just having `free(A)` is not sufficient, since this only frees the `uba` struct itself, and the array inside the struct is not freed. We also cannot do `free(A)` and then `free(A->data)`, because after we free `A`, it is then undefined behavior to dereference `A` to use its fields, since the block of memory it points to has been freed.

```
void uba_free(uba *A) {
  REQUIRES(is_uba(A));
  free(A->data);
  free(A);
}
```

Solution of exercise 2

The first issue is on line 14, with `*area`. When `area` was declared, it was not given a value and is thus uninitialized, and it is undefined behavior to dereference it.

The second issue is line 15, in particular with respect to the call on line 25. On line 25, we call `calc_area` with the address of a local variable, so the pointer we pass in did not come from `malloc/calloc`. Thus it is invalid to free it.

Finally, because we use `xmalloc` on line 27, the fields of `r2` are uninitialized, and any lines which depend on the values of these fields are undefined behavior, meaning line 14 which looks at these fields, as well as line 29 since `area2` is calculated from these uninitialized fields, so we have no idea what value `area2` could contain.

One possible fixed version of the code is as follows:

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

typedef struct rectangle rect;
struct rectangle {
  int x1;
  int y1;
  int x2;
  int y2;
};

int calc_area(rect* r) {
  return (r->x2 - r->x1) * (r->y2 - r->y1);
}

int main() {
  rect r1;
  r1.x1 = 0;
  r1.y1 = 0;
  r1.x2 = 4;
  r1.y2 = 5;
  int area1 = calc_area(&r1);

  rect* r2 = xcalloc(sizeof(rect), 1);
  int area2 = calc_area(r2);
  free(r2);
  if (area2 > area1) return area2;
  return area1;
}
```

Solution of exercise 3

First, we compile our code with the usual flags:

```
% gcc -Wall -Wextra -Werror -Wshadow -std=c99 -pedantic -g file.c
```

It is especially important to have the `-g` flag, since that provides additional debugging information. Then if we run Valgrind with these flags, it will report a number of errors. We will want to look at the very first error reported and fix that, since errors can often cause those which come after them.

Let's run Valgrind and look at the first error:

```
% valgrind --leak-check=full --track-origins=yes ./a.out
...
Use of uninitialised value of size 8
   at 0x10940D: main (file.c:26)
 Uninitialised value was created by a heap allocation
   at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.s
   by 0x109261: xmalloc (xalloc.c:32)
   by 0x1092BC: main (file.c:13)
```

This is telling us that on line 26, we access uninitialized memory, and then that line 13 allocated this uninitialized memory: line 13 of `main` calls `xmalloc`, which then calls `malloc`, but we only care about line 13 which is in our code. This means that some part of our array is not initialized, and after thinking about it for a bit we realize that the first loop actually misses the last element of our array and leaves it uninitialized.

We add the line `arr[len-1] = xcalloc(sizeof(struct point), 1)` to fix this, and run Valgrind again, getting a different error:

```
Invalid read of size 8
   at 0x109431: main (file.c:27)
 Address 0x4a5b238 is 0 bytes after a block of size 504 alloc'd
   at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.s
   by 0x109261: xmalloc (xalloc.c:32)
   by 0x1092BC: main (file.c:13)
```

Here, we are being told that line 27 does an invalid read. The additional context reveals that we are 0 bytes outside of the block allocated on line 13, which is where we allocate our array, so this indicates we are going out of bounds of our array on line 27. Indeed, we can see that doing `arr[i+1]` will go out of bounds, so we change the loop guard to `i < len-1` to stay in bounds.

Now we no longer have any safety issues, but Valgrind still reports some memory leaks. In particular, we get

```
512 (504 direct, 8 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
   at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.s
   by 0x109261: xmalloc (xalloc.c:32)
   by 0x1092BC: main (file.c:13)
```

which tells us that on line 13, we allocate memory which we never free, and indeed we never free `arr`. Let's do that at the bottom the `main` function. However, we still have a memory leak! Valgrind tells us

```
8 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x483DD99: calloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.s
   by 0x10920F: xcalloc (xalloc.c:19)
   by 0x1093F0: main (file.c:23)
```

This says that we are not freeing the memory we allocated for the last element of the array, and indeed the second loop does not free the last element, so we also need to add that in. Once we do this, there are no more errors, and the overall corrected file is as follows:

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

struct point {
  int x;
  int y;
};

int main () {
  int len = 63;
  struct point **arr = xmalloc(sizeof(struct point*) * len);
  for (int i = 0; i < len/2; i++) {
    arr[2*i] = xmalloc(sizeof(struct point));
    arr[2*i + 1] = xmalloc(sizeof(struct point));
    arr[2*i]->x = i+1;
    arr[2*i]->y = i+1;
    arr[2*i+1]->x = i/2;
    arr[2*i+1]->y = 5*i+3;
  }
  arr[len-1] = xcalloc(sizeof(struct point), 1);

  for (int i = 0; i < len-1; i++) {
    if (arr[i]->x + 100 < arr[i+1]->y) {
      printf("%s\n", "Small x value");
    }
    free(arr[i]);
  }
  free(arr[len-1]);
  free(arr);
}
```

Solution of exercise 4 The resulting code is as follows:

```
int main() {
  // ...
  // assume file_t new_text has been previous initialized

  size_t n;

  error_t e = count_words(new_text, &n);
  if (e != 0) printf("count_words failed with error %d.\n", e);
  else printf("The text file contains %zu words", n);

  // ...
}
```

Solution of exercise 5 The updated code is as follows:

```
int main () {
  int  *A = xcalloc(sizeof(int),  100);
  int **B = xcalloc(sizeof(int*), 100);
  int  *C = xcalloc(sizeof(int),  100);

  for (int i = 0; i < 100; i++) {
    A[i] = i*i;
    B[i] = &A[i];
    C[i] = *(B[i]);
  }
  return 0;
}
```

Solution of exercise 6 The updated code is as follows:

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

void mult2(int* x) {
  *x = *x * 2;
}

int main(){
  int* arr = xmalloc(sizeof(int)*5);

  for (int i = 0; i < 5; i++) {
    *(arr + i) = i;
  }
  for (int j = 0; j < 5; j++) {
    mult2(arr + j);
  }

  free(arr);
  return 0;
}
```

Solution of exercise 7 If we make `A` a stack-allocated array instead, it will still have uninitialized values in it until we set the values. Printing everything works the same, but we need to remove the line about freeing `A`, since we can only free pointers returned by `malloc` or `calloc`. As a stack-allocated array, `A` automatically stops existing when the function returns.

Solution of exercise 8 The resulting code is as follows:

```
void swap(struct foobar* x, struct foobar* y) {
  REQUIRES(x != NULL && y != NULL);
  struct foobar temp = *x;
  *x = *y;
  *y = temp;
}
```

This was not possible in C0. In order to swap the values of pointers in C0, we had to work with the fields directly. Variables cannot have struct types in C0, so we could not just swap the structs. In C, however, since C allows for stack allocation of structs, we can make a local copy of the struct. This means that we can make a temporary variable to help with swapping the values.

Solution of exercise 9 Since the shortened string is a prefix of the input string, it is tempting to simply replace the character at position `len` with the `NUL`-terminator, which is what determines where a string ends and therefore its length:

```
char *str_shorten(char *s, size_t len) {
  REQUIRES(len <= strlen(s));
  s[len] = '\0';
  return s;
}
```

This is however incorrect as the input string may be a string literal. String literals reside in the DATA segment of memory and therefore are read-only.

The correct way to solve this problem is to allocate a new string, copy the first `len` characters of `s` into it (the library function `strncpy` does precisely this!), add a `NUL`-terminator, and return the resulting string:

```
char* str_shorten(char *s, size_t len) {
  REQUIRES(len <= strlen(s));
  char* res = xmalloc(sizeof(char) * (len + 1));
  strncpy(res, s, len);
  res[len] = '\0';
  return res;
}
```

Solution of exercise 10 The resulting code is as follows:

```
char *interweave(char *s1, char *s2) {
  size_t len1 = strlen(s1);
  size_t len2 = strlen(s2);

  char *s3 = xmalloc(len1+len2+1);

  size_t i = 0;
  while (i < len1 && i < len2) {
    s3[2*i] = s1[i];
    s3[2*i+1] = s2[i];
    i++;
  }
  size_t remainder = 2*i;
  while (i < len1) {
    s3[remainder] = s1[i];
    remainder++;
    i++;
  }
  while (i < len2) {
    s3[remainder] = s2[i];
    remainder++;
    i++;
  }

  s3[remainder] = '\0';
  return s3;
}
```