

Lecture 8

Data Structures

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

May 27, 2024

In this lecture we introduce the idea of *imperative data structures*. So far, the only interfaces we've used carefully are *pixels* and *string bundles*. Both of these interfaces had the property that, once we created a pixel or a string bundle, we weren't interested in changing its contents. In this lecture, we'll talk about an interface that extends the arrays that are primitively available in C0.

To implement this interface, we'll need to round out our discussion of types in C0 by discussing *pointers* and *structs*, two tastes that go great together. We will discuss using contracts to ensure that pointer accesses are safe.

Additional Resources

- Review slides
 - [Pointers and Structs](https://cs.cmu.edu/~15122/handouts/slides/review/08-pointers.pdf) (<https://cs.cmu.edu/~15122/handouts/slides/review/08-pointers.pdf>)
 - [Libraries](https://cs.cmu.edu/~15122/handouts/slides/review/08-library.pdf) (<https://cs.cmu.edu/~15122/handouts/slides/review/08-library.pdf>)
- [OLI modules](https://cs.cmu.edu/~15122/handouts/oli/08.shtml) (<https://cs.cmu.edu/~15122/handouts/oli/08.shtml>)
- [Code for this lecture](https://cs.cmu.edu/~15122/handouts/code/08-libraries.tgz) (<https://cs.cmu.edu/~15122/handouts/code/08-libraries.tgz>)
- There is one short video associated with this lecture:
 - [Libraries](https://youtu.be/ERGA4PaaUrE) (<https://youtu.be/ERGA4PaaUrE>)

Relating this to our learning goals, we have

Computational Thinking: We illustrate the power of *abstraction* by considering both the client-side and library-side of the interface to a data structure.

Algorithms and Data Structures: The abstract data structure will be one of our first examples of *abstract datatypes*.

Programming: Introduction of structs and pointers, use and design of interfaces.

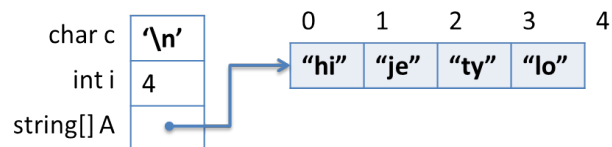
1 Structs

So far in this course, we've worked with five different C0 types — **int**, **bool**, **char**, **string**, and arrays $t[]$ (there is a array type $t[]$ for every type t). The character, Boolean and integer values that we manipulate, store locally, and pass to functions are just the values themselves. For arrays (and strings), the things we store in assignable variables or pass to functions are *addresses*, references to the place where the data stored in the array can be accessed. An array allows us to store and access some number of values of the same type (which we reference as $A[0]$, $A[1]$, and so on).

Therefore, when entering the following commands in Coin (the outputs have been elided),

```
--> char c = '\n';
--> int i = 4;
--> string[] A = alloc_array(string, 4);
--> A[0] = "hi";
--> A[1] = "je";
--> A[2] = "ty";
--> A[3] = "lo";
```

the interpreter will store something like the following in its memory:



The next data structure we will consider is the struct. A *struct* can be used to aggregate together different types of data, which helps us create data structures. By contrast, an array is an aggregate of elements of the *same* type.

Structs must be explicitly declared in order to define their “shape”. For example, if we think of an image, we want to store an array of pixels alongside the width and height of the image, and a struct allows us to do that:

```

struct img_header {
    pixel_t[] data;
    int width;
    int height;
};

```

Here `data`, `width`, and `height` are *fields* of the struct. The declaration expresses that every image has an array of `pixel_t`s called `data` as well as a `width` and a `height`. This description is incomplete, as there are some missing consistency checks — we would expect the length of `data` to be equal to the `width` times the `height`, for instance, but we can capture such properties in a separate data structure invariant.

C0 values such as integers, characters, the address of an array are *small*. Depending on the computer, an address is either 64 bits long or 32 bits long, which means that the *small types* take at most 64 bits to represent. Because structs can have multiple components, they can grow too large for the computer to easily copy around, and C0 does not allow us to use structs as locals:

```

% coin structs.c0
C0 interpreter (coin) 0.3.2 'Nickel'
Type '#help' for help or '#quit' to exit.
--> struct img_header IMG;
<stdio>:1.1-1.22:error:type struct img_header not small
[Hint: cannot pass or store structs in variables directly; use
pointers]

```

Therefore, we can only create structs in allocated memory, just like we can only store the contents of arrays in allocated memory. (This is true even if they happen to be small enough to fit into 32 bits.) Instead of **alloc_array** we call **alloc** which returns a *pointer* to the struct that has been allocated in memory. Let's look at an example in `coin`.

```

--> struct img_header* IMG = alloc(struct img_header);
IMG is 0xFFAFF20 (struct img_header*)

```

We can access the fields of a struct, for reading or writing, through the notation `p->f` where `p` is a pointer to a struct, and `f` is the name of a field in that struct. Continuing above, let's see what the default values are in the allocated memory.

```

--> IMG->data;
(default empty int[] with 0 elements)
--> IMG->width;

```

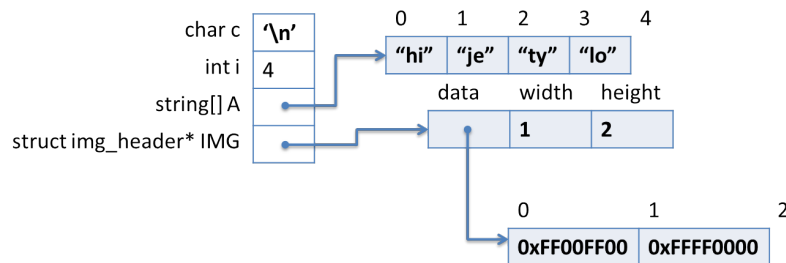
```
0 (int)
--> IMG->height;
0 (int)
```

We can write to the fields of a struct by using the arrow notation on the left-hand side of an assignment.

```
--> IMG->data = alloc_array(pixel_t, 2);
IMG->data is 0xFFAFC130 (int[] with 2 elements)
--> IMG->width = 1;
IMG->width is 1 (int)
--> (*IMG).height = 2;
(*IMG).height is 2 (int)
--> IMG->data[0] = 0xFF00FF00;
IMG->data[0] is -16711936 (int)
--> IMG->data[1] = 0xFFFF0000;
IMG->data[1] is -65536 (int)
```

The notation `(*p).f` is a longer form of `p->f`. First, `*p` follows the pointer to arrive at the struct in memory, then `.f` selects the field `f`. We will rarely use this dot-notation `(*p).f` in this course, preferring the arrow-notation `p->f`.

An updated picture of memory, taking into account the initialization above, looks like this:



2 Pointers

As we have seen in the previous section, a pointer is needed to refer to a struct that has been created in allocated memory. It can also be used more generally to refer to an element of arbitrary type that has been created in allocated memory. For example:

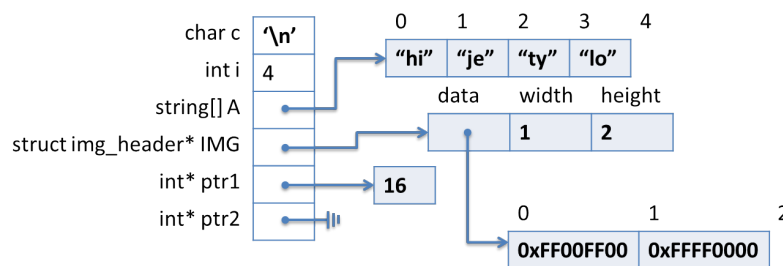
```
--> int* ptr1 = alloc(int);
ptr1 is 0xFFAFC120 (int*)
--> *ptr1 = 16;
*(ptr1) is 16 (int)
--> *ptr1;
16 (int)
```

In this case, we refer to the value of `p` using the notation `*p`, either to read (when we use it inside an expression) or to write (if we use it on the left-hand side of an assignment).

So we would be tempted to say that a pointer value is simply an address. But this story, which was correct for arrays, is not quite correct for pointers. There is also a special value `NULL`. Its main feature is that `NULL` is not a valid address, so we cannot dereference it to obtain stored data. For example:

```
--> int* ptr2 = NULL;
ptr2 is NULL (int*)
--> *ptr2;
Error: null pointer was accessed
Last position: <stdio>:1.1-1.3
```

Graphically, `NULL` is sometimes represented with the ground symbol, so we can represent our updated setting like this:



To rephrase, we say that a pointer value is an address, of which there are two kinds. A valid address is one that has been allocated explicitly with `alloc`, while `NULL` is an invalid address. In C, there are opportunities to create many other invalid addresses, as we will discuss in another lecture.

Attempting to dereference the null pointer is a safety violation in the same class as trying to access an array with an out-of-bounds index. In C0, you will reliably get an error message, but in C the result is undefined and will not necessarily lead to an error. Therefore:

*Whenever you dereference a pointer p , either as $*p$ or $p->f$, you must have a reason to know that p cannot be $NULL$.*

In many cases this may require function preconditions or loop invariants, just as for array accesses.

3 Creating an interface

See the short video on [Libraries](https://www.youtube.com/embed/ERGA4PaaUrE) at <https://www.youtube.com/embed/ERGA4PaaUrE>.

The next ten lectures for this class will focus on building, analyzing, and using different data structures. When we're thinking about implementing data structures, we will almost always use pointers to structs as the core of our implementation.

In the rest of this lecture, we will build a data structure for *self-sorting arrays*. Like arrays, they will contain a fixed number of elements of a given type that can be accessed through an index (we will limit ourselves to self-sorting arrays of strings). Unlike C0 arrays, the values in a self-sorting array are sorted, and remain so as we update its values. That allows a programmer using our self-sorting arrays to look for elements using binary search, in $O(\log n)$ time if n is the number of contained elements.

Self-sorting arrays work mostly like arrays of strings. The primitive operations that C0 provides on string arrays are the ability to create a new array, to get a particular index of an array, and to set a particular index in an array. We capture these as three functions that act on an abstract type `ssa_t` (mnemonic for self-sorting array type):

```
// typedef _____ ssa_t;
ssa_t ssa_new(int size);      // ~ alloc_array(string, size)
string ssa_get(ssa_t A, int i);    // ~ A[i]
void ssa_set(ssa_t A, int i, string x); // ~ A[i] = x
```

But this is not a complete picture! An interface needs to also capture the preconditions necessary for using that abstract type *safely*. For instance, we know that safety of array access requires that we only create non-negative-length arrays and we never try to access a negative element of an array:

```
ssa_t ssa_new(int size)      /*@requires size >= 0 @*/;
string ssa_get(ssa_t A, int i) /*@requires 0 <= i; @*/;
```

This still isn't enough: our contracts need to ensure an upper bound so that we don't access the element at index 100 of a length-12 array. We don't have the `\length()` method, as it is a primitive for C0 arrays, not our new `ssa_t` type. So we need an additional function in our interface to get the length, and we'll use that in our contracts.

```
int ssa_len(ssa_t A);
string ssa_get(ssa_t A, int i)
    /*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

It's important to emphasize what just happened. Because we want the type `ssa_t` to be *abstract*, we can't use `\length` in a contract: we can only use

`\length` for arrays. Because we have to be able to write a contract that explains how to use the data type safely, we need to extend our interface with a new function `ssa_len`. But because this function is in the interface, the client can access the length of the array — something that can't be done for C0 arrays (outside of a contract)! So we *do* know something about `ssa_t` now: it can't just be `string[]`, because if it was there would be no way to implement `ssa_len`.

For this reason, we're going to say that `ssa_t` is not an unknown type but that it is an unknown *pointer* type. The updated pseudo-typedef below shows how we indicate this:

```
// typedef _____* ssa_t;

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

ssa_t ssa_new(int size)
/*@requires 0 <= size;           @*/
/*@ensures \result != NULL;     @*/
/*@ensures ssa_len(\result) == size; @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;           @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;

void ssa_set(ssa_t A, int i, string x)
/*@requires A != NULL;           @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

Admitting that `ssa_t` is a pointer also means that we have to add a lot of NULL checks to the interface — as the client of the `ssa_t` type, we know that a value of this type is either a valid pointer to the self-sorting array data structure, or it is NULL — and we disallow NULL as the representation of *any* valid self-sorting array.

4 The Library Perspective

When we implement the library for `ssa_t`, we will declare a type `ssa` as a synonym for `struct ssa_header`, which has a `length` field to hold the length and a `data` field to hold the actual array.

```
struct ssa_header { // Implementation type
```

```

    int length;
    string[] data;
};
typedef struct ssa_header ssa;    // Abbreviation

typedef ssa* ssa_t;              // Interface type

```

The last line is where we make a connection between the interface type `ssa_t` exported to the user by the interface, and the type `ssa` (or equivalently `struct ssa_header`) the library uses to implement the exported functionalities. As promised, it is a pointer type. We usually write it as the very last line of the implementation.

Inside the library implementation, we'll use `ssa*` instead of `ssa_t` to emphasize that we're manipulating a pointer structure. Outside, we use exclusively `ssa_t` as our abstract type of supped up arrays. Using this knowledge, we can begin to implement the array interface from the library side, though we immediately run into safety issues.

```

int ssa_len(ssa* A)
//@requires A != NULL;
{
    return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
{
    return A->data[i];
}

```

In both cases, the precondition `A != NULL` allows us to say that the dereferences `A->length` and `A->data` are safe. But how do we know `A->data[i]` is not an out-of-bounds array access? We don't — the second precondition of `ssa_get` just tells us that `i` is non-negative and less than whatever `ssa_len` returns!

If we want to use the knowledge that `ssa_len(A)` returns the length of `A->data`, then we'd need to add `\result == \length(A->data)` as a postcondition of `ssa_len`...

...and we can only prove that postcondition true if we add the precondition `A->length == \length(A->data)` to `ssa_len`...

...and if we do that, it changes the safety requirements for the call to `ssa_len` in the preconditions of `ssa_get`, so we also have to add the precondition `A->length == \length(A->data)` to `ssa_get`.

The user, remember, didn't need to know anything about this, because they were ignorant of the internal implementation details of the `ssa_t` type. As long as the user respects the interface, only creating `ssa_t`'s with `ssa_new` and only manipulating them with `ssa_len`, `ssa_get`, and `ssa_set`, they should be able to expect that the contracts on the interface are sufficient to ensure safety. But we don't have this luxury from the library perspective: all the functions in the library's implementation are going to depend on all the parts of the data structure making sense with respect to all the other parts. We'll capture this notion in a new kind of invariant, a *data structure invariant*.

4.1 Data Structure Invariants

We can apply operational reasoning as library designers to say that, as long as the `length` field of an `ssa` is set correctly by `ssa_new`, it must remain correct throughout all calls to `ssa_get` and `ssa_set`. But, as with operational reasoning about loops, this is an error-prone way of thinking about our data structures. Our solution in this case will be to capture what we know about the well-formedness of an array in an invariant; we expect that any `ssa` being handled by the user will satisfy this data structure invariant.

The above invariants for self-sorting arrays are pretty simple: a `ssa` is well-formed if it is a non-NULL pointer to a struct where `\length(A->data) == A->length`. They capture safety but do not say anything about the array being sorted. We will add a correctness invariant, that the array be in fact sorted. This is achieved through a function `ssa_sorted` which is implemented similarly to the corresponding function for arrays of integers — it can be found in the code accompanying this lecture. If we try to turn this into a mathematical statement that captures the overall well-formedness requirements for our implementation, we get the *specification function* `is_ssa`:

```
bool is_ssa(ssa* A) {
    return A != NULL
        && is_array_expected_length(A->data, A->length)
        && ssa_sorted(A);
}
```

While we would like `is_array_expected_length` to be a function that returns `true` when the given array has the expected length and `false` otherwise, the restriction of length-checking to contracts makes this impossible to write in C0. In this one case, we'll allow ourselves to write a data structure invariant that might raise an assertion error instead of returning `false`:

```
bool is_array_expected_length(string[] A, int length) {
```

```
    //@assert \length(A) == length;
    return true;
}
```

Whenever possible, however, we prefer data structure invariants that return `true` or `false` to data structure invariants that raise assertion failures.

The data structure invariant, then, implies the postcondition of `ssa_len`, and so the function `ssa_get` will require the data structure invariant to hold as well, satisfying the precondition of `ssa_len`.

```
int ssa_len(ssa* A)
//@requires is_ssa(A);
//@ensures \result == \length(A->data);
{
    return A->length;
}
```

```
string ssa_get(ssa* A, int i)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
{
    return A->data[i];
}
```

Functions that create new instances of the data structure should ensure that the data structure invariants hold of their result, and functions that modify data structures should have postconditions to ensure that none of those data structure invariants have been violated.

```
ssa* ssa_new(int size)
//@requires 0 <= size;
//@ensures is_ssa(\result);
{
    ssa* A = alloc(ssa);
    A->length = size;
    A->data = alloc_array(string, size);
    return A;
}
```

```
void ssa_set(ssa* A, int i, string x)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
//@ensures is_ssa(A);
{
```

```

A->data[i] = x;

// Move x up the array if needed
for (int j=i; j < A->length-1 &&
     string_compare(A->data[j],A->data[j+1]) > 0;
     j++)
    //@loop_invariant i <= j && j <= A->length - 1;
    swap(A->data, j, j+1);

// Move x down the array if needed
for (int j=i; j > 0 &&
     string_compare(A->data[j],A->data[j-1]) < 0;
     j--)
    //@loop_invariant 0 <= j && j <= i;
    swap(A->data, j, j-1);
}

```

The code below `A->data[i] = x` moves `x` up or down the array `A->data` to ensure that the sortedness invariant remains satisfied once the function returns. As a consequence, it is unlikely that a subsequent call to `ssa_get(A, i)` will return `x` as its value. This differs from C0 arrays, for which elements can always be found where we just put them.

Now that we have added data structure invariants, our operational reasoning for why `ssa_get` was safe can be formalized as an invariant. Any client that respects the interface will only ever get and will only ever manipulate arrays that satisfy the data structure invariants, so we know that the data structure invariants we're counting on for safety will hold at runtime.

Invariants Aren't Usually Part of the Interface

When we have interfaces that hide implementations from the user, then the data structure invariant, captured here by the function `is_ssa`, should *not* be a part of the interface. Clients don't need to know that the internal invariants are satisfied; as long as they're using `ssa` according to the interface, their invariants should be satisfied.

This applies to all aspects of `is_ssa`, in particular to the correctness invariant checked by the function `ssa_sorted`. But how shall the client be sure that the elements of a self-sorting array are and remains sorted as it is used? The client may trust the developer of the data structure library. Another option is to write a client-side version of `ssa_sorted` using *only the operations exported by the interface*.

4.2 Debugging a Library Implementation

The implementation of the functions `ssa_len` and `ssa_get`, and even `ssa_new`, is pretty straightforward. But `ssa_set` is more involved since it moves elements around to keep the array sorted. Having a bug in our first version is not unlikely. How to notice our code has a bug? And once we know there is a bug, how to find where it is so that we can fix it?

There are two complementary approaches to noticing that our code contains a bug. The first is to have a good reason to believe that potentially unsafe operations are actually safe (array accesses out of bounds in our example). As we did earlier, using contracts to *reason* about our code is a great way to do so: in this way we can identify and fix errors even before we try to run our code the first time. However, some bugs are quite subtle and we may miss them when reasoning about our program. This is when the second approach comes into play: *testing*. Once we have a program we are relatively confident about, we want to write a battery of *test cases* to identify any lingering bugs. Except sometimes for very large test cases, we always want to run out tests in debug mode (by compiling them with the `-d` flag).

Once testing reveals a bug, we need to find out exactly where it is so that we can fix it. Sometimes, this is quite easy: a simple inspection of the function that fails the test will do it. Often, however, bugs are not as obvious. There are two main ways to identify a bug. The first one relies on contracts, in particular data structure invariants. When a test fails because of a contract violation, the execution will abort by reporting the exact line where the contract failed. From there, it is usually easy to find what went wrong and to fix it... except when what fails is a data structure invariant.

The second approach to figuring out what is causing a bug is to use *print statements*. One particularly useful form of print statements is one that prints the contents of the data structure the library is implementing. To this end, we write a *print function* that we can use in our implementation to inspect the contents of the data structure anytime we feel something may be off.

In fact, we recommend writing *two* data structure print functions: one that does not check the representation invariant and one that does. An *unsafe print function*, which does not check the presentation invariant, is useful when the data structure invariant function fails (for example if we had forgotten to sort the array when writing `ssa_set` or if we had written it incorrectly so that it fails on valid self-sorting arrays). Here is one such function for our example:

```
void ssa_print_unsafe(ssa* A) {  
    int len = A->length;  
    printf("SSA length: %d; SSA data: [", len);
```

```

for (int i = 0; i < len; i++)
  //@loop_invariant 0 <= i && i <= len;
  {
    printf(A->data[i]);
    if (i < len-1) printf(", ");
  }
  printf("]");
}

```

If the bug does not cause the data structure invariant to fail (for example if our code for `ssa_set` had written the new string `x` in every position in the array `A->data`) or if our contracts are not strong enough (i.e., our data structure invariant function accepts invalid self-sorting array), we are better off using a print function that does check the data structure invariant:

```

void ssa_print_internal(ssa* A)
  //@requires is_ssa(A);
  {
    ssa_print_unsafe(A);
  }

```

Like the specification function `is_ssa`, the print functions live in the library implementation and are not exported to the client. (Some libraries may want to export a print function, but it will often be different in order to hide implementation details.) Like `is_ssa`, it is good practice to write `ssa_print_unsafe` and `ssa_print_internal` before implementing any library function. Notice that `ssa_print_unsafe` uses the fields of the underlying implementation type rather than the functions exported by the interface (e.g., `A->length` as opposed to `ssa_len(A)`). This is because we want to use it to catch bugs in these very functions! (We sometimes relax this rule for library functions that are particularly simple.)

5 The Client Perspective

The client of our self-sorting array library can use any function exported by the library (plus the type `ssa_t`) *and nothing else*. As an example, we write a client-side printing function.

```

void display_this_ssa(ssa_t A)
  //@requires A != NULL;
  {
    printf("\n");
    for (int j = 0; j < ssa_len(A); j++)
      //@loop_invariant 0 <= j;
  }

```

```
{
    printf("%d => %s\n", j, ssa_get(A, j));
}
}
```

Notice that this code uses the functions exported by the interface (here `ssa_len` and `ssa_get`). Using the fields `length` and `data` of its implementation would violate the interface, and most likely result in bugs when compiling this code with another implementation of this library. Similarly, defining `display_this_ssa` as

```
void display_this_ssa(ssa_t A)
//@requires A != NULL;
{
    ssa_print_internal(A);
}
```

would be a violation of the interface because `ssa_print_internal` is not exported by the library.

6 Exercises

Exercise 1 (sample solution on page 21). You want to keep track of your friends' contact information, namely their name, phone number, email, and address. A good way to do so is to use a struct. An address has itself many components: a street name, a numerical zip code, a city, and a state — that's another struct!

Define a struct to allow you to record a person's contact information and one that records their address. Since it is tedious to write out entire struct name, use **typedef** to give these structs a short nickname.

Then, write a main function that simply creates and populates the contact info for a friend or any entity.

Exercise 2 (sample solution on page 22). Consider the following C0 program

```
int main() {
    int* w = alloc(int);
    int i = 122;
    int** x = alloc(int*);
    // Memory diagram 1

    *x = w;
    *w = 15;
    w = alloc(int);
    int* y = alloc(int);
    int*** z = alloc(int**);
    // Memory diagram 2

    **x = 9;
    *z = x;
    **z = w;
    ***z = i + 5 - *y;
    *y = *w - 3;
    // Memory diagram 3

    return 0;
}
```

Draw the state of both the local and the allocated memory when execution reaches lines 5, 12, and 19.

Exercise 3 (sample solution on page 24). Using point-to reasoning, justify the safety of each line of the following code snippet, both for pointer and array operations. If a line does not contain operations that could be unsafe, write N/A.

If you need to point to multiple lines to justify the safety of a line, enter the line numbers separated by a comma.

```

1 int main() {
2   int* p = alloc(int);
3   *p = 7;
4   int*[] A = alloc_array(int*, 2);
5   A[0] = p;
6   A[1] = alloc(int);
7   *A[1] = 5;
8   int*[]* B = alloc(int*[]);
9   *B = alloc_array(int*, 2);
10  (*B)[0] = A[1];
11  (*B)[1] = p;
12
13  return 0;
14 }
```

Exercise 4 (sample solution on page 24). In homework, we implemented a small pixel library where pixels were defined to be C0 integers. Another option is to use a struct, with one field for each component of the pixel, as follows:

```

struct pixel_as_struct {                               // Concrete implementation type
  int alpha;     // 0 <= alpha < 256
  int red;       // 0 <= red < 256
  int green;     // 0 <= green < 256
  int blue;      // 0 <= blue < 256
};
typedef struct pixel_as_struct pixel; // Concrete implementation type
```

Using this definition of the type `pixel`, implement the following interface for pixels:

```

// typedef _____ pixel_t;           // Abstract client type

pixel_t make_pixel(int alpha, int red, int green, int blue)
/*@requires 0 <= alpha && alpha < 256; @*/
/*@requires 0 <= red && red < 256; @*/
/*@requires 0 <= green && green < 256; @*/
/*@requires 0 <= blue && blue < 256; @*/ ;

int get_alpha(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;
```

```
int get_red(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;
```

```
int get_green(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;
```

```
int get_blue(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;
```

For compatibility with an implementation that defines `pixel_t` as `int`, this interface does not check that the pixels passed to and returned from the exported functions are not `NULL`.

Note that, as part of this task, you will need to define a representation function `is_pixel` and use it as you implement the functions exported by the interface.

Exercise 5 (sample solution on page 26). We want to extend the `pixels` library in the previous exercise with functions `set_alpha`, `set_red`, `set_green` and `set_blue` that allow updating the value of a pixel component. For example, the call `set_green(p, 122)` will return a new pixel that differs from `p` only by the value of the green component now being 122. In subsequent uses of `p`, the green component will have its original value.

Write the interface prototypes of these four functions. Include contracts that are as strict as possible.

Then, implement these function as part of a library implementation that uses the type definition for pixels as the previous exercise — although it is possible do not implement them as client functions of this library. Include all necessary implementation-side contracts. You may assume that any function you have written as part of the previous exercise is available to you.

Exercise 6 (sample solution on page 29). Let's expand the previous two exercises further. You tell a friend about your new struct-based implementation of the pixel library and your extension using the `set_` functions. Impressed, they ask you if they can use your new library as part of their own image-manipulation application. You agree. A few days later, your friend shows you this client function they used based on your library.

```
void complement_pixel(pixel* p)
/*@requires p != NULL;
  @ensures is_pixel(p);
{
    p->red = 255 - p->red;
    p->green = 255 - p->green;
    p->blue = 255 - p->blue;
}
```

This code contains several mistakes. What are they? Once you have identified them, write a correct client-side version of this same function.

Sample Solutions

Solution of exercise 1 Because the address is part of the contact information, we shall define a struct for the address first. We call it **struct** `address_info` and typedef it to `address`. At this point, we do the same for the contact information, declaring the type **struct** `contact_info` and abbreviating it as `contact`.

```
struct address_info {
    string street;
    int zipcode;
    string city;
    string state;
};
typedef struct address_info address;

struct contact_info {
    string name;
    string phone;
    string email;
    address* address;
};
typedef struct contact_info contact;
```

A few observations are in order:

- Because the type of addresses is a struct, we will want to define the address field in **struct** `contact_info` as a pointer type.
- It make sense for most of the fields to be defined as strings. We use an integer instead for the zip code because zip codes fit comfortably in a 32-bit C0 **int**. Since US phone numbers are 10 digits long, several would not fit in 32 bits. For this reason, we defined the phone number field as a string.
- Note that we used the same name (`address`) both for as a field name and as a type name. This is perfectly legal in C0, but we will generally refrain from doing so to avoid confusion.

The following main function creates CMU's contact information:

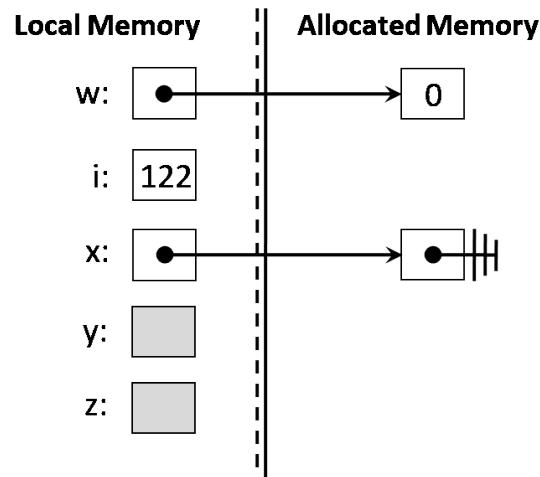
```
int main() {
    address* cmuadr = alloc(address);
    cmuadr->street = "5000 Forbes Avenue";
    cmuadr->zipcode = 15213;
    cmuadr->city = "Pittsburgh";
    cmuadr->state = "PA";

    contact* cmu = alloc(contact);
    cmu->name = "Carnegie Mellon University";
    cmu->phone = "412-268-2000";
    cmu->email = "info@cmu.edu";
    cmu->address = cmuadr;

    return 0;
}
```

Solution of exercise 2 Here's the state of memory when execution reaches each of the three lines.

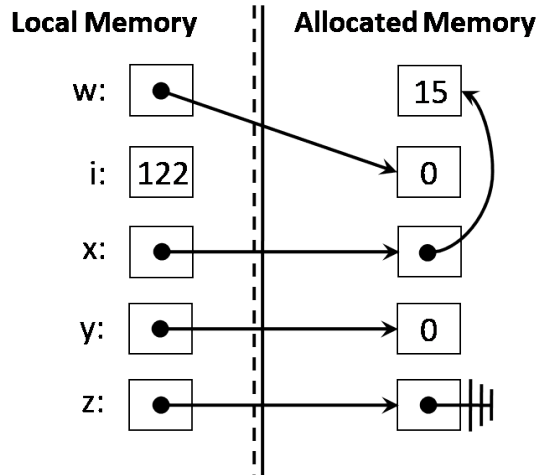
Line 5 Memory diagram 1



At this point of the execution, the memory cells for variables `w` and `x` have been allocated and initialized to the default value of their type, `0` for `w` since it points to a cell that can contain values of type `int`, and `NULL` for `x` as it points to a cell that can contain values of pointer type (`int*` here).

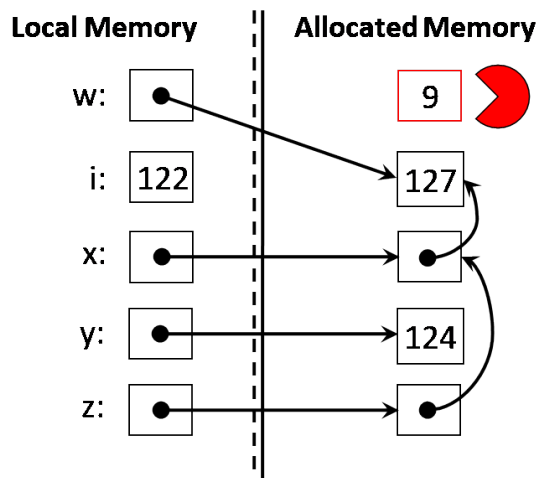
The variables `y` and `z` have not been initialized yet, and are therefore grayed out.

Line 12 Memory diagram 2



By the time the execution reaches this line, the cell referenced by **x** points to the cell that **w** was pointing to in the previous memory snapshot (the cell containing 15), but **w** now contains the address of a new cell in memory whose initial value is the default **int**, i.e., 0. Memory has also been allocated for the variables **y** and **z**. These cells have been initialized to 0 (the default **int**) and NULL (the default **int****), respectively.

Line 19 Memory diagram 3



At the end of the program, the cell originally pointed to by **w** reachable only as ***x** in the previous diagram (and containing 15) has had

its value changed to 9. The cell pointed to by `z` has been set to point to the cell pointed to by `x`, and the contents of this cell (reached as `**z`) is repointed to the same cell that `w` points to. This change causes the cell containing 9 not to be accessible from any variable in local memory, which means that it will eventually be reclaimed by the garbage collector. The last two lines of code change the values of some of these cell by following pointers.

Solution of exercise 3 Here is the justification for each line:

2. N/A
3. is safe by line 2
4. N/A
5. is safe by line 4
6. is safe by line 4
7. is safe by lines 4 and 6
8. N/A
9. is safe by line 8
10. is safe by lines 8, 9, and 4
11. is safe by lines 8 and 9

Solution of exercise 4 Here is the implementation part of this pixel library.


```
bool is_pixel(pixel* p) {
    return p != NULL
        && (0 <= p->alpha && p->alpha < 256)
        && (0 <= p->red && p->red < 256)
        && (0 <= p->blue && p->blue < 256)
        && (0 <= p->green && p->green < 256);
}

pixel* make_pixel(int alpha, int red, int green, int blue)
//@requires 0 <= alpha && alpha < 256;
//@requires 0 <= red && red < 256;
//@requires 0 <= green && green < 256;
//@requires 0 <= blue && blue < 256;
//@ensures is_pixel(\result);
{
    pixel* p = alloc(pixel);
    p->alpha = alpha;
    p->red = red;
    p->green = green;
    p->blue = blue;
    return p;
}

int get_alpha(pixel* p)
//@requires is_pixel(p);
//@ensures 0 <= \result && \result < 256;
{
    return p->alpha;
}

int get_red(pixel* p)
//@requires is_pixel(p);
//@ensures 0 <= \result && \result < 256;
{
    return p->red;
}
```

```
int get_green(pixel* p)
//@requires is_pixel(p);
//@ensures 0 <= \result && \result < 256;
{
    return p->green;
}
```

```
int get_blue(pixel* p)
//@requires is_pixel(p);
//@ensures 0 <= \result && \result < 256;
{
    return p->blue;
}
```

```
typedef pixel* pixel_t;           // Abstract client type
```

The data representation invariant function `is_pixel` checks that its argument is not `NULL` (since `NULL` is not a valid pixel) and that the value of each component of the pixel is in the range $[0, 256)$.

The function `make_pixel` allocates a new `pixel` and populates its fields with the values it received in input before returning it. Each `get_` function returns the value of the appropriate field.

In addition to the contracts mentioned in the interface, the function `make_pixel` ensures as a postcondition that the returned pixel is valid, which it does using `is_pixel`. The four `get_` functions expect a valid pixel, which they requires again using `is_pixel`.

The implementation ends with the definition of the abstract type `pixel_t` as `pixel*`.

Solution of exercise 5 The interface declarations of the `set_` functions is as follows:

```
pixel_t set_alpha(pixel_t p, int alpha)
/*@requires 0 <= alpha && alpha < 256;      @*/
/*@ensures get_alpha(\result) == alpha;      @*/
/*@ensures get_red(\result) == get_red(p);   @*/
/*@ensures get_green(\result) == get_green(p); @*/
/*@ensures get_blue(\result) == get_blue(p); @*/ ;

pixel_t set_red(pixel_t p, int red)
/*@requires 0 <= red && red < 256;          @*/
/*@ensures get_alpha(\result) == get_alpha(p); @*/
/*@ensures get_red(\result) == red;          @*/
/*@ensures get_green(\result) == get_green(p); @*/
/*@ensures get_blue(\result) == get_blue(p); @*/ ;

pixel_t set_green(pixel_t p, int green)
/*@requires 0 <= green && green < 256;      @*/
/*@ensures get_alpha(\result) == get_alpha(p); @*/
/*@ensures get_red(\result) == get_red(p);   @*/
/*@ensures get_green(\result) == green;      @*/
/*@ensures get_blue(\result) == get_blue(p); @*/ ;

pixel_t set_blue(pixel_t p, int blue)
/*@requires 0 <= blue && blue < 256;        @*/
/*@ensures get_alpha(\result) == get_alpha(p); @*/
/*@ensures get_red(\result) == get_red(p);   @*/
/*@ensures get_green(\result) == get_green(p); @*/
/*@ensures get_blue(\result) == blue;        @*/ ;
```

Each checks that the component value it takes a input is in the range $[0, 256)$. Each checks as a postcondition that the corresponding `get_` function returns the a pixel with that very component value equal to what we passed in while the other components remain unchanged. Again, we do not check that the pixels itself is non-NULL because we want this interface to work with implementations that define `pixel_t` as the type `int`.

Because we want to be able to define `pixel_t` as `int`, we cannot make these into `void` functions: were we to update the pixel `p` itself (rather than returning a new pixel), the change would not be visible to caller of `pixel_t` is `int`.

The simplest way to implement these functions is to use `make_pixel` to create a new pixel with the change component from the input and the unchanged components from `p`:

```
pixel* set_alpha(pixel* p, int alpha)
//@requires is_pixel(p);
//@requires 0 <= alpha && alpha < 256;
//@ensures is_pixel(\result);
//@ensures get_alpha(\result) == alpha;
//@ensures get_red(\result) == get_red(p);
//@ensures get_green(\result) == get_green(p);
//@ensures get_blue(\result) == get_blue(p);
{
    return make_pixel(alpha, get_red(p), get_green(p), get_blue(p));
}
```

```
pixel* set_red(pixel* p, int red)
//@requires is_pixel(p);
//@requires 0 <= red && red < 256;
//@ensures is_pixel(\result);
//@ensures get_alpha(\result) == get_alpha(p);
//@ensures get_red(\result) == red;
//@ensures get_green(\result) == get_green(p);
//@ensures get_blue(\result) == get_blue(p);
{
    return make_pixel(get_alpha(p), red, get_green(p), get_blue(p));
}
```

```
pixel* set_green(pixel* p, int green)
//@requires is_pixel(p);
//@requires 0 <= green && green < 256;
//@ensures is_pixel(\result);
//@ensures get_alpha(\result) == get_alpha(p);
//@ensures get_red(\result) == get_red(p);
//@ensures get_green(\result) == green;
//@ensures get_blue(\result) == get_blue(p);
{
    return make_pixel(get_alpha(p), get_red(p), green, get_blue(p));
}
```

```
pixel* set_blue(pixel* p, int blue)
//@requires is_pixel(p);
//@requires 0 <= blue && blue < 256;
//@ensures is_pixel(\result);
//@ensures get_alpha(\result) == get_alpha(p);
//@ensures get_red(\result) == get_red(p);
//@ensures get_green(\result) == get_green(p);
//@ensures get_blue(\result) == blue;
{
    return make_pixel(get_alpha(p), get_red(p), get_green(p), blue);
}
```

Of note here are the implementation-side contracts: naturally, we check that the input pixel is valid using the precondition `is_pixel(p)`, but we also want to be sure that the returned pixel is valid as well. We do so by having `is_pixel(\result)` also as a postcondition.

For practice, you may want to check that this postcondition is provable using point-to reasoning.

Solution of exercise 6 The function `complement_pixel`, as written by your friend, breaks the library interface by using implementation details. This is bad because this code is unlikely to work, or even compile, were your to swap your implementation of the extended pixel library for a different one. Specifically,

- The type `pixel` is internal to the implementation. A client-side function should use exclusively the abstract type `pixel_t`.
- The precondition `p != NULL` assumes pixels are pointer types, but this is mentioned nowhere in the interface.
- The postcondition makes use of the data structure invariant function `is_pixel`, but this again is internal to the implementation and should not be used in client code.
- The body of the function assumes pixels are implemented as structs with fields `red`, `green` and `blue`, but once more the interface says nothing of this sort. This code relies on implementation details.

An equivalent client-side implementation of `complement_pixel` is as follows.

```
pixel_t complement_pixel(pixel_t p) {
    int r = 255 - get_red(p);
    int g = 255 - get_green(p);
    int b = 255 - get_blue(p);
    return make_pixel(get_alpha(p), r, g, b);
}
```

It only uses the functions exported by the interface of the extended pixel library. We could add contracts to make sure that the components of the returned pixel are what we expect.