Lecture Notes on
Pointers

15-122: Principles of Imperative Computation
Frank Pfenning, Rob Simmons

Lecture 10
September 28, 2013

1 Introduction

In this lecture we complete our discussion of types in C0 by discussing *pointers* and *structs*, two great tastes that go great together. We will discuss using contracts to ensure that pointer accesses are safe, as well as the use of *linked lists* to implement the stack and queue interfaces that were introduced last time. The linked list implementation of stacks and queues allows us to handle lists of any length.

Relating this to our learning goals, we have

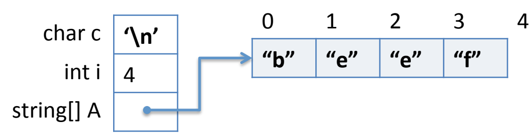
Computational Thinking: We emphasize the importance of *abstraction* by producing a second implementation of the stacks and queues we introduced in the last lecture.

Algorithms and Data Structures: *Linked lists* are a fundamental data structure.

Programming: We will see *structs* and *pointers*, and the use of recursion in the definition of structs.

2 Structs and pointers

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, string, Boolean, and integer values that we manipulate, store locally, and pass to functions are just the values themselves; the picture we work with looks like this:



When we consider arrays, 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.

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 to create data structures. In 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:

```
typedef int pixel;

struct img_header {
  pixel[] data;
  int width;
  int height;
};
```

Here $data$, $width$, and $height$ are not variables, but *fields* of the struct. The declaration expresses that every image has an array of $data$ 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.

Structs do not necessarily fit into a machine word because they can have arbitrarily many components, so they must be allocated on the heap (in memory, just like arrays). This is true even if they happen to be small enough to fit into a word (in order to maintain a uniform and simple language implementation).

```
% coin structdemo.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]
```

How, then, do we manipulate structs? We use the same solution as for arrays: we manipulate them via their address in memory. 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 0xFFAFFF20 (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.

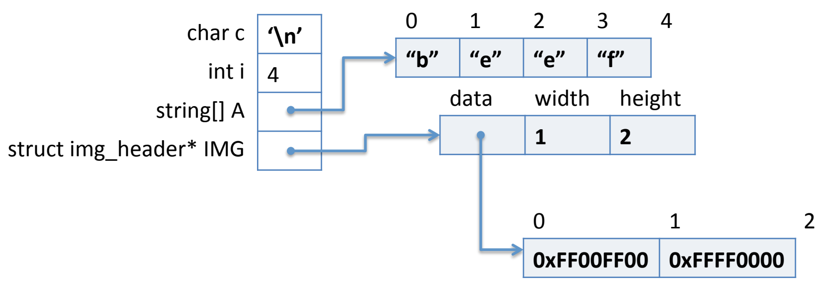
```
--> IMG->data;
(default empty int[] with 0 elements)
--> IMG->width;
0 (int)
--> IMG->height;
0 (int)
```

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, 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:



3 Pointers

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

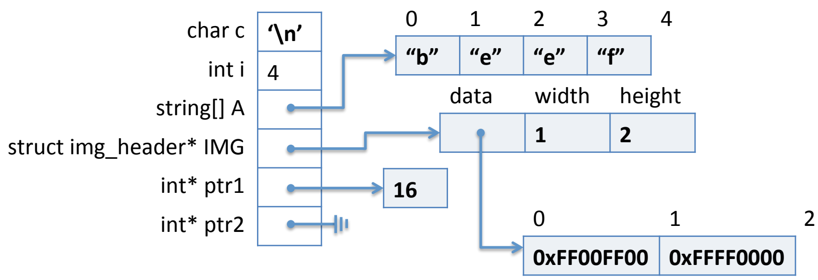
```
--> int* ptr1 = alloc(int);
ptr1 is 0xFFAFC120 (int*)
--> *ptr1 = 16;
*(ptr1) is 16 (int)
--> *ptr1;
16 (int)
```

In this case we refer to the value 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;
p 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.

4 Linked Lists

Linked lists are a common alternative to arrays in the implementation of data structures. Each item in a linked list contains a data element of some type and a *pointer* to the next item in the list. It is easy to insert and delete elements in a linked list, which are not natural operations on arrays, since arrays have a fixed size. On the other hand access to an element in the middle of the list is usually $O(n)$, where n is the length of the list.

An item in a linked list consists of a struct containing the data element and a pointer to another linked list. In C0 we have to commit to the type of element that is stored in the linked list. We will refer to this data as having type `elem`, with the expectation that there will be a type definition elsewhere telling C0 what `elem` is supposed to be. Keeping this in mind ensures that none of the code actually depends on what type is chosen. These considerations give rise to the following definition:

```
struct list_node {
  elem data;
  struct list_node* next;
};
typedef struct list_node list;
```

This definition is an example of a *recursive type*. A struct of this type contains a pointer to another struct of the same type, and so on. We usually use the special element of type `t*`, namely `NULL`, to indicate that we have reached the end of the list. Sometimes (as will be the case for our use of linked lists in stacks and queues), we can avoid the explicit use of `NULL` and obtain more elegant code. The type definition is there to create the type name `list`, which stands for `struct list_node`, so that a pointer to a list node will be `list*`.

There are some restriction on recursive types. For example, a declaration such as

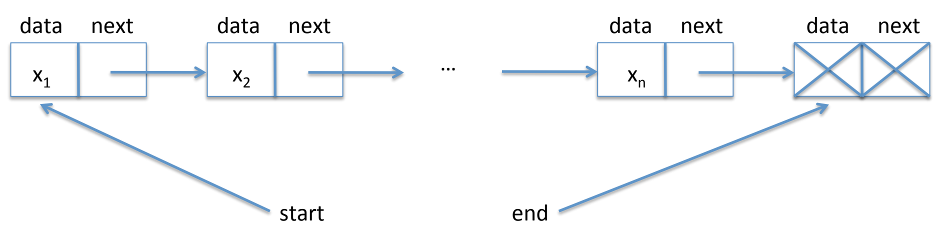
```
struct infinite {
  int x;
  struct infinite next;
}
```

would be rejected by the C0 compiler because it would require an infinite amount of space. The general rule is that a struct can be recursive, but the recursion must occur beneath a pointer or array type, whose values are addresses. This allows a finite representation for values of the struct type.

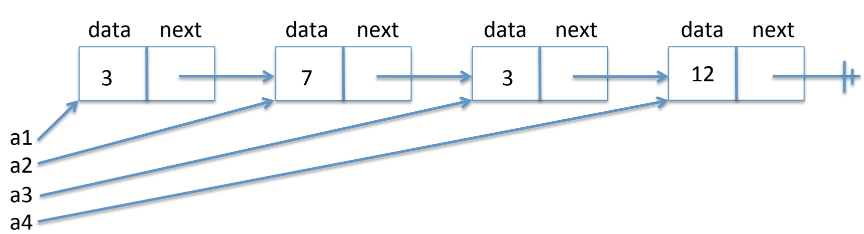
We don't introduce any general operations on lists; let's wait and see what we need where they are used. Linked lists as we use them here are a *concrete type* which means we do *not* construct an interface and a layer of abstraction around them. When we use them we know about and exploit their precise internal structure. This is contrast to *abstract types* such as queues or stacks (see next lecture) whose implementation is hidden behind an interface, exporting only certain operations. This limits what clients can do, but it allows the author of a library to improve its implementation without having to worry about breaking client code. Concrete types are cast into concrete once and for all.

5 List segments

A lot of the operations we'll perform in the next few lectures are on *segments* of lists: a series of nodes starting at *start* and ending at *end*.



This is the familiar structure of an "inclusive-lower, exclusive-upper" bound: we want to talk about the data in a series of nodes, ignoring the data in the last node. That means that, for any non-NULL list node pointer `l`, a segment from l to l is empty (contains no data). Consider the following structure:



According to our definition of segments, the data in the segment from *a1* to *a4* is the sequence $3, 7, 3$, the data in the segment from *a2* to *a3* contains the sequence 7, and the data in the segment from *a1* to *a1* is the empty sequence. Note that if we compare the pointers *a1* and *a3* C0 will tell us they are *not equal* – even though they contain the same data they are different locations in memory.

Given an inclusive beginning point *start* and an exclusive ending point *end*, how can we check whether we have a segment from *start* to *end*? The simple idea is to follow *next* pointers forward from *start* until we reach *end*. If we reach NULL instead of *end* then we know that we missed our desired endpoint, so that we do not have a segment. (We also have to make sure that we say that we do not have a segment if either *start* or *end* is NULL, as that is not allowed by our definition of segments above.) We can implement this simple idea in all sorts of ways:

Recursively

```
bool is_segment(list* start, list* end) {
  if (start == NULL) return false;
  if (start == end) return true;
  return is_segment(start->next, end);
}
```

For loop

```
bool is_segment(list* start, list* end) {
  for (list* p = start; p != NULL; p = p->next) {
    if (p == end) return true;
  }
  return false;
}
```

While loop

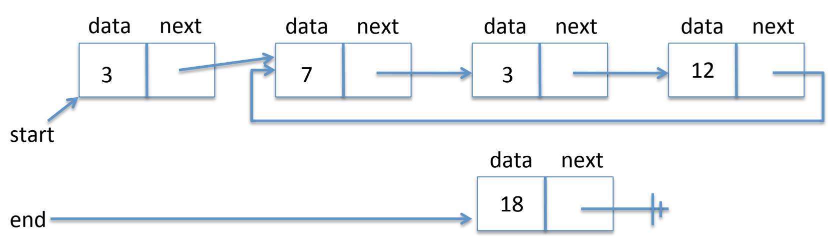
```
bool is_segment(list* start, list* end) {
  list l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

However, every one of these implementations of `is_segment` has the same problem: if given a circular linked-list structure, the specification function `is_segment` may not terminate.

It's quite possible to create structures like this, intentionally or unintentionally. Here's how we could create the above structure in Coin:

```
--> list* start = alloc(list);
--> start->data = 3;
--> start->next = alloc(list);
--> start->next->data = 7;
--> start->next->next = alloc(list);
--> start->next->next->data = 3;
--> start->next->next->next = alloc(list);
--> start->next->next->next->data = 12;
--> start->next->next->next->next = start->next;
--> list* end = alloc(list);
--> end->data = 18;
--> end->next = NULL;
--> is_segment(start, end);
```

and this is what it would look like:



While it is not strictly necessary, *whenever possible, our specification functions should return* `true` *or* `false` *rather than not terminating or raising an assertion violation*. We do treat it as strictly necessary that our specification functions should always be safe – they should never divide by zero, access an array out of bounds, or dereference a null pointer. We will see how to address this problem in our next lecture.

6 Checking for circularity

In the 9am lecture, after a couple of false starts we developed a reasonable way of checking for circularity. The idea that is implicit in the solution we discovered is that, if you have a circular linked list structure, then eventually you are going encounter a node that you've seen before. So what we need is an always-terminating way of checking all the places we've already seen.

A helper function that can be useful for this operation is one that we'll call `is_in_bounded(x, start, n)`. This function tells us whether we'll get to the list node `x` from `start` by following no more than `n` pointers.

```
bool is_in_bounded(list* x, list* start, int numsteps)
//@requires 0 <= numsteps;
{
  int i = 0;
  for (list* p = start; p != NULL; p = p->next)
    //@loop_invariant 0 <= i && i <= numsteps;
    {
      if (i == numsteps) {
        // If the thing we're looking for is in the list,
        // it is further on.
        return false;
      }

      if (p == x) {
        // Oh, here it is!
        return true;
      }

      i += 1;
    }

  // We reached a NULL, it's not in the bounded list
  return false;
}
```

On the i^{th} iteration of our naive `is_segment` loop, we know that we can get from `start` to `p` by following exactly i pointers. We know we have a cycle if we can *also* get `p` by following *fewer* than i pointers. In our example of a circular linked list, we could get to the list node containing 7 by following either 1 `next` pointer or 4 `next` pointers, and we will return `false` the second time we encounter that node.

```
// Always returns true or false
bool is_segment(list* start, list* end) {
  int i = 0;
  for (list* p = start; p != NULL; p = p->next)
    //@loop_invariant 0 <= i;
    {
      //@assert(is_in_bounded(p, start, i+1));
      if (is_in_bounded(p, start, i)) return false; // CYCLE!
      if (p == end) return true; // DONE!
      i += 1;
    }

  // We reached NULL without getting to end first
  return false;
}
```

Exercises

Exercise 1 *We say "on the* ith *iteration of our naive* `is_segment` *loop, we know that we can get from* `start` *to* `p` *by following exactly* i *pointers." Write a function* `is_reachable_in(list* start, list* end, int numsteps)`*; this function should return* `true` *if we can get from* `start` *to* `end` *in exactly* `numsteps` *steps. Use this function as a loop invariant for* `is_segment`.