# Lecture Notes on Pointers

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

Lecture 9
February 14, 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 ensure 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.

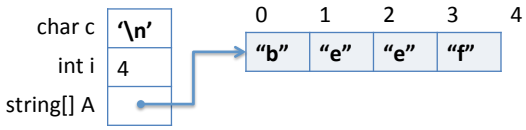
**Algorithms and Data Structures:** *Linked lists* are a fundamental data structure.

**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, but 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. The picture we work with looks like this:




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. Compare this to arrays, which is an aggregate of elements of the same type.

Structs must be explicitly declared. 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 if they happen to be small enough to fit into a word in order to maintain a uniform and simple language.

```
% 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 structs, 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.

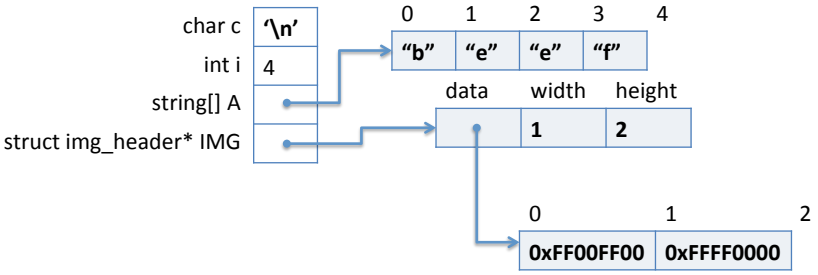
```
--> IMG->data;
(default empty int[] with 0 elements)
--> IMG->width;
0 (int)
--> IMG->height;
0 (int)
```

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:

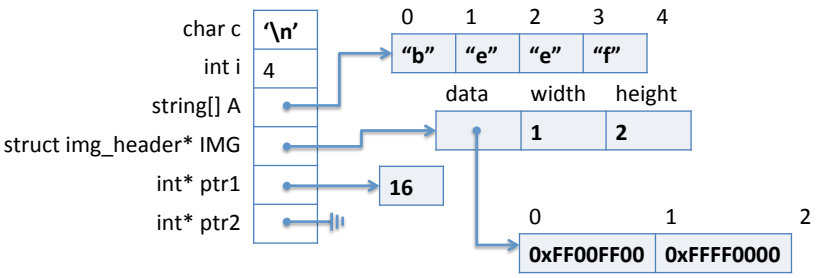
```
--> int* ptr1 = alloc(int);
ptr1 is 0xFFAFC120 (int*)
--> *ptr1 = 16;
*(ptr1) is 16 (int)
--> *ptr1;
16 (int)
```

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;
}
```

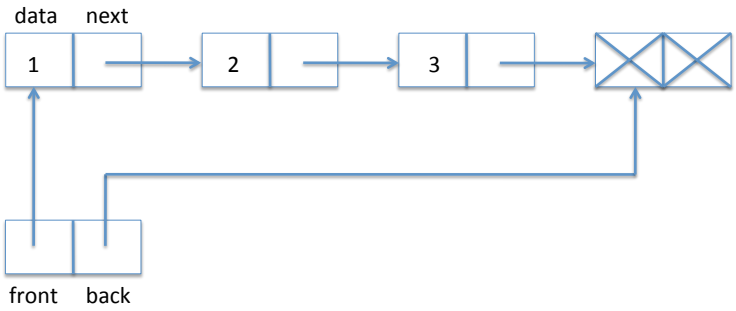
would be rejected by the C0 compiler because it would require an infinite amount of space. The general rule is that a struct can be recursive, but the recursion must occur beneath a pointer or array type, whose values are addresses. This allows a finite representation for values of the struct type.

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 Queues with Linked Lists

Here is a picture of the queue data structure the way we envision imple-




A queue is implemented as a struct with a `front` and `back` field. The `front` field points to the front of the queue, the `back` field points to the back of the queue. We need these two pointers so we can efficiently access both ends of the queue, which is necessary since dequeue (front) and enqueue (back) access different ends of the list.

In the array implementation of queues, we kept the back as one greater than the index of the last element in the array. In the linked-list implementation of queues, we use a similar strategy, making sure the `back` pointer points to one element past the end of the queue. Unlike arrays, there must be something in memory for the pointer to refer to, so there is always one extra element at the end of the queue which does not have valid data or `next` pointer. We have indicated this in the diagram by writing `X`.

The above picture yields the following definition.

```
struct queue_header {
  list* front;
  list* back;
};
typedef struct queue_header* queue;
```

We call this a *header* because it doesn't hold any elements of the queue, just pointers to the linked list that really holds them. The type definition allows us to use `queue` as a type that represents a *pointer to a queue header*. We define it this way so we can hide the true implementation of queues from the client and just call it an element of type `queue`.

When does a struct of this type represent a valid queue? In fact, whenever we define a new data type representation we should first think about the data structure invariants. Making these explicit is important as we think about and write the pre- and postconditions for functions that implement the interface.

What we need here is if we follow `front` and then move down the linked list we eventually arrive at `back`. We call this a *list segment*. We also want both `front` and `back` not to be `NULL` so it conforms to the picture, with one element already allocated even if the queue is empty.

```
bool is_queue(queue Q) {
  if (Q == NULL) return false;
  if (Q->front == NULL) return false;
  if (Q->back == NULL) return false;
  if (!is_segment(Q->front, Q->back)) return false;
  return true;
}
```

Next, the code for checking whether two pointers delineate a list segment. When both `start` and `end` are `NULL`, we consider it a valid list segment, even though this will never come up for queues. It is a common code pattern for working with linked lists and similar data representation to have a pointer variable, here called $p$, that is updated to the next item in the list on each iteration until we hit the end of the list.

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

Here we stop in two situations: if $p =$ `NULL`, then we cannot come up against $end$ any more because we have reached the end of the list and we

return `false`. The other situation is if we find $end$, in which case we return `true` since we have a valid list segment. This function may not terminate if the list contains a cycle. We will address this issue in the next lecture; for now we assume all lists are acyclic.

To check if the queue is empty we just compare its front and back. If they are equal, the queue is empty; otherwise it is not. We require that we are being passed a valid queue. Generally, when working with a data structure, we should always require and ensure that its invariants are satisfied in the pre- and post-conditions of the functions that manipulate it. Inside the function, we will generally temporarily violate the invariants.

```
bool queue_empty(queue Q)
//@requires is_queue(Q);
{
  return Q->front == Q->back;
}
```

To obtain a new empty queue, we just allocate a list struct and point both front and back of the new queue to this struct. We do not initialize the list element because its contents are irrelevant, according to our representation. It is good practice to always initialize memory if we care about its contents, even if it happens to be the same as the default value placed there.

```
queue queue_new()
//@ensures is_queue(\result);
//@ensures queue_empty(\result);
{
  queue Q = alloc(struct queue_header);
  list* p = alloc(struct list_node);
  Q->front = p;
  Q->back = p;
  return Q;
}
```

Let's take one of these lines apart. Why does

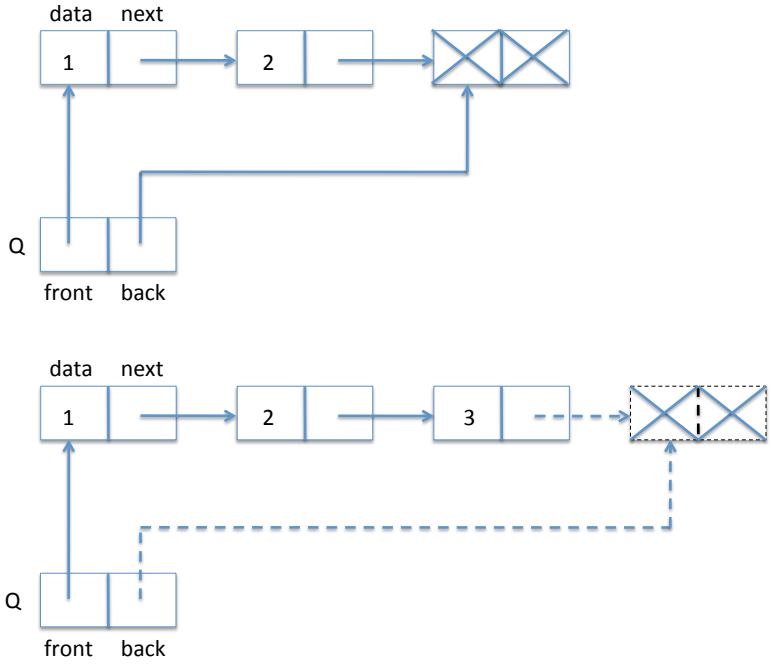
```
  queue Q = alloc(struct queue_header);
```

make sense? According to the definition of `alloc`, we might expect

```
  struct queue_header* Q = alloc(struct queue_header);
```

since allocation returns the address of what we allocated. Fortunately, we defined `queue` to be a short-hand for `struct queue_header*` so all is well.

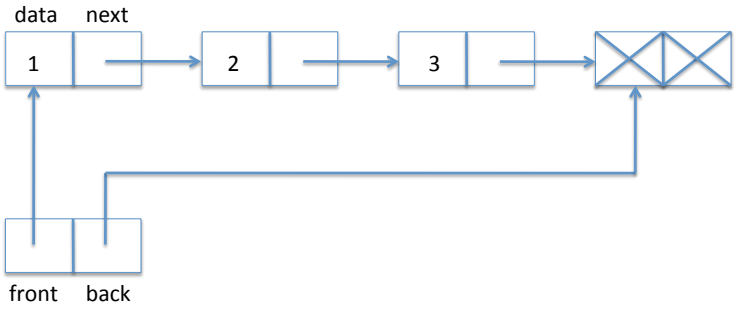
To enqueue something, that is, add a new item to the back of the queue, we just write the data (here: a string) into the extra element at the back, create a new back element, and make sure the pointers updated correctly. You should draw yourself a diagram before you write this kind of code. Here is a before-and-after diagram for inserting "3" into a list. The new or updated items are dashed in the second diagram.

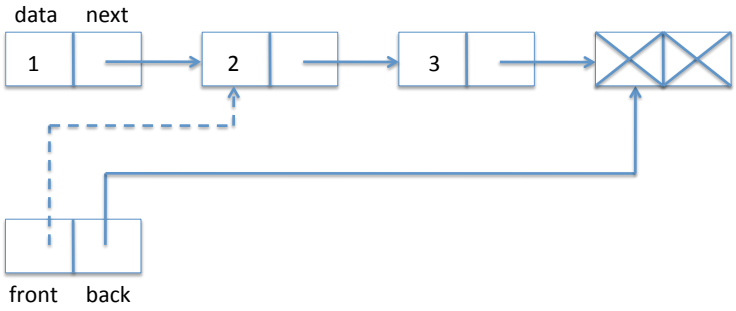



Another important point to keep in mind as you are writing code that manipulates pointers is to make sure you perform the operations in the right order, if necessary saving information in temporary variables.

```
void enq(queue Q, string s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{
  list* p = alloc(struct list);
  Q->back->data = s;
  Q->back->next = p;
  Q->back = p;
}
```

Finally, we have the dequeue operation. For that, we only need to change the front pointer, but first we have to save the dequeued element in a temporary variable so we can return it later. In diagrams:




And in code:

```
string deq(queue Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
  string s = Q->front->data;
  Q->front = Q->front->next;
  return s;
}
```

Let's verify that the our pointer dereferencing operations are safe. We have

```
Q->front->data
```

which entails two pointer dereference. We know `is_queue(Q)` from the precondition of the function. Recall:

```
bool is_queue(queue Q) {
  if (Q == NULL) return false;
  if (Q->front == NULL) return false;
  if (Q->back == NULL) return false;
  if (!is_segment(Q->front, Q->back)) return false;
  return true;
}
```

We see that `Q->front` is okay, because by the first test we know that `Q != NULL` is the precondition holds. By the second test we see that both `Q->front` and `Q->back` are not null, and we can therefore dereference them.

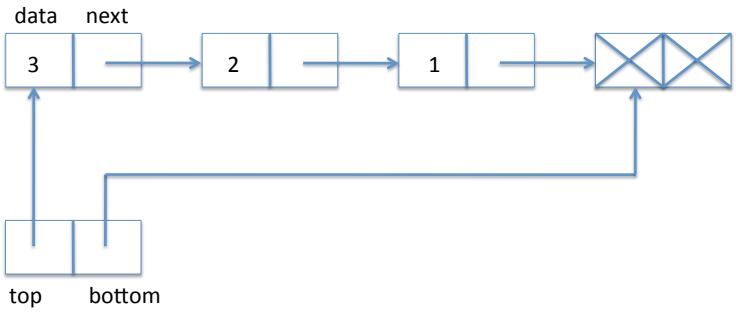
We also make the assignment `Q->front = Q->front->next`. Why does this preserve the invariant? Because we know that the queue is not empty (second precondition of `deq`) and therefore `Q->front != Q->back`. Because `Q->front` to `Q->back` is a valid non-empty segment, `Q->front->next` cannot be null.

An interesting point about the dequeue operation is that we do not explicitly deallocate the first element. If the interface is respected there cannot be another pointer to the item at the front of the queue, so it becomes *unreachable*: no operation of the remainder of the running programming could ever refer to it. This means that the garbage collector of the C0 runtime system will recycle this list item when it runs short of space.

## 6 Stacks with Linked Lists

For the implementation of stacks, we can reuse linked lists and the basic structure of our queue implementation, except that we read off elements from the same end that we write them to. We call the pointer to this end `top`. Since we do not perform operations on the other side of the stack, we do not necessarily need a pointer to the other end. For structural reasons, and in order to identify the similarities with the queue implementation, we still decide to remember a pointer `bottom` to the bottom of the stack. With this design decision, we do not have to handle the bottom of the stack much different than any other element on the stack. The difference is that

the data at the bottom of the stack is meaningless and will not be used in our implementation. A typical stack then has the following form:




Here, 3 is the element at the top of the stack.

We define:

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

struct stack_header {
  list* top;
  list* bottom;
};
typedef struct stack_header* stack;
```

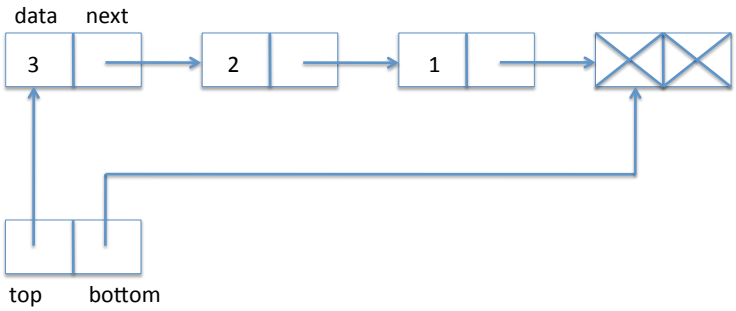
To test if some structure is a valid stack, we only need to check that the list starting at `top` ends in `bottom`; this is almost identical to the data structure invariant for queues:

```
bool is_stack(stack S) {
  if (S == NULL) return false;
  if (Q->front == NULL) return false;
  if (Q->back == NULL) return false;
  if (!is_segment(Q->front, Q->back)) return false;
  return true;
}
```

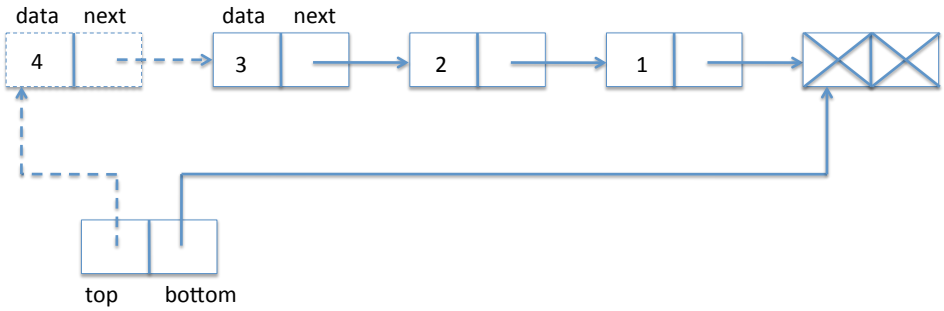
Popping from a stack requires taking an item from the front of the linked list, which is much like dequeuing.

```
elem pop(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
  elem e = S->top->data;
  S->top = S->top->next;
  return e;
}
```

To push an element onto the stack, we create a new list item, set its data field and then its next field to the current top of the stack – the opposite end of the linked list from the queue. Finally, we need to update the `top` field of the stack to point to the new list item. While this is simple, it is still a good idea to draw a diagram. We go from




to



In code:

```
void push(stack S, elem e)
//@requires is_stack(S);
//@ensures is_stack(S);
{
  list* p = alloc(struct list_node);
  p->data = e;
  p->next = S->top;
  S->top = p;
}
```

This completes the implementation of stacks, which are a very simple and pervasive data structure.

## Exercises

**Exercise 1** *Consider what would happen if we* `pop` *an element from the empty stack when contracts are not checked in the linked list implementation? When does an error arise?*

**Exercise 2** *Stacks are usually implemented with just one pointer in the header, to the top of the stack. Rewrite the implementation in this style, dispensing with the* `bottom` *pointer, terminating the list with* `NULL` *instead.*