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 – \texttt{int}, \texttt{bool}, \texttt{char}, \texttt{string}, and arrays \texttt{t[]} (there is a array type \texttt{t[]} for every type \texttt{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 \texttt{A[0]}, \texttt{A[1]}, and so on.

The next data structure we will consider is the \textit{struct}. A \textit{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 along-side the width and height of the image, and a struct allows us to do that:

\begin{verbatim}
typedef int pixel;

struct img_header {
    pixel[] data;
    int width;
    int height;
};
\end{verbatim}

Here \texttt{data}, \texttt{width}, and \texttt{height} are not variables, but \textit{fields} of the struct. The declaration expresses that every image has an array of \texttt{data} as well as a \texttt{width} and a \texttt{height}. This description is incomplete, as there are some missing consistency checks – we would expect the length of \texttt{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.
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;

3 Pointers

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:

---

LECTURE NOTES
*(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.