15-122: Principles of Imperative Computation

Recitation 9 — Josh Zimmerman

Interface vs. Implementation and Abstraction

An *interface* is a set of functions (and possibly variables) that are exposed to people who use a library. For instance, for a stack we have the interface functions: `bool stack_empty(stack S);`, `stack stack_new();`, `void push(stack S, elem e);`, `elem pop(stack S);`. Someone who uses this library only needs to specify what type `elem` is and use these functions. They don't need to worry about how the stack is implemented: it could use a linked list, an array, or some other data structure.

An *implementation* of an interface is the actual code and data structures used to write it. We wrote a relatively simple (but not particularly useful) implementation of stacks in lecture. In the implementation, we had to specify how everything works (we had to write the functions and define `structs`) so that we can actually use the data structure.

The difference between interface and implementation is critical. If I'm using some stack implementation, I don't want to think about *how* the author wrote it. I just want to be able to assume that it works and does what I want it to do. If the author changes the way the implementation of the stack works, my code should still work without any modifications (assuming that the stack code is still correct). This makes my code far more maintainable and reliable—it doesn't matter what version of the stack library I'm using or even who wrote it. If any implementation conforms to that interface then my code should work with it.

This has some important consequences: namely, if some variable or function is not in an interface, you *SHOULD NOT* use it. If you do and then the author of the library changes their code, yours will break in a way that might be incredibly difficult to detect and fix.

For example, in lecture on Thursday we'll change the implementation of stacks to make them work better. If a user of the stack library relied on the way we implemented stacks yesterday, they'd find that their code breaks when the implementation of stacks is improved.

Another key idea that comes up when talking about interface and implementations is the idea of *abstraction*. When we talk about an interface, we're *abstracting away* the detail of how that interface is implemented. Instead, we talk about how we can use that interface.

A lot of implementations are very complicated, so this idea of *abstraction* is critical to writing software: it lets us write the stack code once and then not have to remember how it works when we're writing code that uses it.

Typedefs

It's often annoying to type out the full name of a type, either because it's a long name, because it's difficult to conceptually think about the original name, or because we want to make our code work with elements of multiple types.

For example, if you want to store `int`s in your stack, you might use the command `typedef int elem;`. Then, when you compile your code with the stack implementation, the compiler will realize that everywhere the author of the stack library wrote "`elem`", it should fill in "`int`".

On assignment 1, some of you may have noticed that `imageutil.c0` had the line `typedef int pixel;` in it. In this typedef, we're saying to the compiler "everywhere I write `pixel` in my code, I really mean `int`.

The reason we had this typedef is that we wanted you to think about `pixels` just in terms of the underlying bits rather than as numbers that you can do arithmetic with. In this case the new type name is longer, but using a `typedef` allows you more easily think about the problems we asked you to solve rather than the underlying implementation.

One helpful way to remember the order of the type names for `typedefs` is that if you remove the word `typedef`, you'd be declaring a variable. For instance, if we write `int pixel;` we're declaring an `int` and we're calling it `pixel`. Adding the `typedef` just changes this from a variable declaration to a type declaration.

Stacks and Queues

In lecture, we talked about stacks (a last in, first out data structure) and queues (a first in, first out data structure).

In other words, you can insert items into both stacks and queues, but the order that they come out in is different. In a queue, elements come out in exactly the same order that they go in: If I insert 1, 2, and 3 into a queue (in that order) and then dequeue an element, I'll get 1.

Stacks work differently: If I insert 1, 2, and 3 into a stack (in that order) and then pop an element, I'll get 3.

We'll go through a few examples of stacks and queues now, on the board.

`structs`

A *struct* is a way of putting a bunch of things together and referring to them with names. It's really convenient to put a bunch of variables together when we're implementing queues and stacks, as well as other data structures. Here's some of the syntax (we'll talk about more details on Friday).

```
1  struct point {
2     int x;
3     int y;
4  };
5
6  struct point* add_points (struct point *p1, struct point *p2)
7  //@requires p1 != NULL && p2 != NULL;
8  //@ensures \result != NULL;
9  {
10    struct point *new_point = alloc(struct point);
11    new_point -> x = p1 -> x + p2 -> x;
12    new_point -> y = p1 -> y + p2 -> y;
13    return new_point;
14 }
15 int main () {
16    struct point *p1 = alloc(struct point);
17    p1 -> x = 1;
18    p1 -> y = 0;
19    struct point *p2 = alloc(struct point);
20    p2 -> x = 3;
21    p2 -> y = 1;
22
23    struct point *sum = add_points(p1, p2);
24    //@assert sum -> x == 4;
25    //@assert sum -> y == 1;
26    return 0;
27 }
```

clac

clac is a relatively simple *postfix*-based programming language. As we read in numbers from the input (which we represent as a queue), we push operands onto a stack and act on them based on the instructions that are in the queue.

Here's an example of *clac* processing some input (you can get this yourself when working on claclab.

```
$ clac-ref -trace
Clac top level
clac>> 5 9 2 7 3 + - / dup * %
                                  stack || queue
                                        || 5 9 2 7 3 + - / dup * %
                                      5 || 9 2 7 3 + - / dup * %
                                    5 9 || 2 7 3 + - / dup * %
                                  5 9 2 || 7 3 + - / dup * %
                                5 9 2 7 || 3 + - / dup * %
                              5 9 2 7 3 || + - / dup * %
                               5 9 2 10 || - / dup * %
                                 5 9 -8 || / dup * %
                                   5 -1 || dup * %
                                 5 -1 -1 || * %
                                     5 1 || %
                                       0 ||
0
```

What's happening here? Well, we push all of the numbers onto the stack after reading them out of the queue. Then, we get to the +, so we pop two items (the 7 and the 3) off of the stack, add them, and push their sum, 10, back on. Next, we get to the -, pop off the 2 and 10 and subtract them, and get -8, which we push on to the stack. Then, we get to the /. We pop 9 and -8 and divide them. 9/-8 rounds to -1, so we push that onto the stack. Next, we execute the `dup`, which simply makes the top element of the stack appear twice. We get to the *, which multiplies the top two elements, giving us `1`. Finally, we get to the %. `5 % 1 == 0`, so we push 0. Then, we're out of instructions, so we end and pop the top item off of the stack and print it.

A common source of confusion with `clac` is `if` statements and `else` statements.

When we get to an `if` statement, we pop the top item off of the stack. If it is 0, we skip the next *two* tokens in the queue – we just ignore them. Otherwise (if it's non-zero), we continue processing tokens as normal.

When we get to an `else` statement, we *always* skip the next token in the queue.

So, why are these if/else statements? Let's take a look at some `clac` code

NOTE: Whenever I type `x` in clac code below, I'm using it to mean any arbitrary int – you should fill in an int, like 1, -1, 0, etc, if you're actually running the code.

```
$ clac-ref -trace
Clac top level
clac>> 0 if 2 else 3
                                  stack || queue
                                        || 0 if 2 else 3
                                      0 || if 2 else 3
                                        || 3
                                      3 ||
3
clac>> 1 if 2 else 3
                                  stack || queue
                                      3 || 1 if 2 else 3
                                    3 1 || if 2 else 3
                                        || 2 else 3
                                    3 2 || else 3
                                    3 2 ||
2
```

Next, let's write a simple `clac` program: one that calculates absolute value. We can define $|x|$ as follows:

$$|x| = \begin{cases} x * 1 & \text{if } x \geq 0 \\ x * -1 & \text{if } x < 0 \end{cases}$$

So, if x is less than 0, we want to multiply it by -1 and otherwise we want to multiply it by 1. If we run the `clac` command `x 0 <`, then it will result in `1` being on the top of the stack if $x < 0$ and `0` being on the top of the stack otherwise.

We eventually want to multiply by either 1 or -1, so we should push the appropriate one of them onto the stack: If $x < 0$ we multiply by -1, otherwise we multiply by 1.

So, we add `if -1 else 1` to our command. Now we have

```
x 0 < if -1 else 1
```

This says "if $x < 0$, push -1 onto the stack. Otherwise, push 1 onto the stack." This works because when `x 0 <` evaluates to `0` (so $x \geq 0$), we ignore the tokens `-1` and `else`, so we just push `1` onto the stack. If `x 0 <` evaluates to `1` (so $x < 0$), then we push `-1` onto the stack and ignore the token `1`.

Next, we want to multiply by `x`, so we add `*` to the end:

```
x 0 < if -1 else 1 *
```

This doesn't work, though! We popped `x` off of the stack when we did the comparison. If we run the above command, we get:

```
Error: Error: not enough elements on stack
```

So, we need to duplicate `x` before we compare, so we can still use it later:

```
x dup 0 < if -1 else 1 *
```

That will compute the absolute value of `x`. Now we can define a function to do this for us!

```
 : abs dup 0 < if -1 else 1 * ;
```

Now, if I type `x abs` into `clac`, I'll get the absolute value of `x` on the top of the stack (if `x` is an `int`).