filename: c_5p_function-pointers_20220603.txt
https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)FunctionPointers.html

C/FunctionPointers

Note: You are looking at a static copy of the former PineWiki site, used for class notes by James Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions of some of these notes at http://www.cs.yale.edu/homes/aspnes/#classes.

1. Basics

A function pointer, internally, is just the numerical address for the code for a function. When a function name is used by itself without parentheses, the value is a pointer to the function, just as the name of an array by itself is a pointer to its zeroth element. Function pointers can be stored in variables, structs, unions, and arrays and passed to and from functions just like any other pointer type. They can also be called: a variable of type function pointer can be used in place of a function name.

2. Function pointer declarations

A function pointer declaration looks like a function declaration, except that the function name is wrapped in parentheses and preceded by an asterisk. For example:

```
/* a function taking two int arguments and returning an int */
int function(int x, int y);

/* a pointer to such a function */
int (*pointer)(int x, int y);
```

As with function declarations, the names of the arguments can be omitted.

Here's a short program that uses function pointers:

```
 /* Functional "hello world" program */

 #include <stdio.h>

 int
 main(int argc, char **argv) {
     /* function for emitting text */
     int (*say)(const char *);

     say = puts;
     say("hello world");

     return 0;
}
```

3. Applications

Function pointers are not used as much in C as in functional languages, but there are many common uses even in C code.

3.1. Callbacks

The classic example is qsort, from the standard library:

```
/* defined in stdlib.h */
void qsort(void *base, size_t n, size_t size,
int (*cmp)(const void *key1, const void *key2));
```

This is a generic sorting routine that will sort any array in place. It needs to know (a) the base address of the array; (b) how many elements there are; (c) how big each element is; and (d) how to compare two elements. The only tricky part is supplying the comparison, which could involve arbitrarily-complex code. So we supply this code as a function with an interface similar to strcmp.

```
static int compare_ints(void *key1, void *key2) {
    return *((int *) key1) - *((int *) key2);
}

int sort_int_array(int *a, int n) {
    qsort(a, n, sizeof(*a), compare_ints);
}
```

Other examples might include things like registering an error handler for a library, instead of just having it call abort() or something equally catastrophic, or providing a cleanup function for freeing data passed into a data structure.

3.2. Dispatch tables

Alternative to gigantic if/else if or switch statements. See page 234 of [https://www.cs.yale.edu/homes/aspnes/pinewiki/KernighanPike.html]KernighanPike for a good example of this.

3.3. Iterators

See [**1]C/Iterators.

4. Closures

A closure is a function plus some associated state. A simple way to implement closures in C is to use a static local variable, but then you only get one. Better is to allocate the state somewhere and

pass it around with the function. For example, here's a simple functional implementation of infinite sequences, that generalizes the example in [**2]AbstractDataTypes:

```
/* a sequence is an object that returns a new value each time it is called */
struct sequence {
    int (*next)(void *data);
    void *data;
};

typedef struct sequence *Sequence;

Sequence create_sequence(int (*next)(void *data), void *data) {
    Sequence s;

    s = malloc(sizeof(*s));
    assert(s);

    s->next = next;
    s->data = data;

    return s;
}

int sequence_next(Sequence s) {
    return s->next(s->data);
}
```

And here are some examples of sequences:

```
/* make a constant sequence that always returns x */
static int constant_sequence_next(void *data) {
    return *((int *) data);
}

Sequence constant_sequence(int x) {
    int *data;

    data = malloc(sizeof(*data));
    if(data == 0) return 0;

    *data = x;

    return create_sequence(constant_sequence_next, data);
}

/* make a sequence x, x+a, x+2*a, x+3*a, ... */
struct arithmetic_sequence_data {
    int cur;
    int step;
};

static int arithmetic_sequence_next(void *data) {
    struct arithmetic_sequence_data *d;

    d = data;
    d->cur += d->step;

    return d->cur;
}

Sequence arithmetic_sequence(int x, int a) {
    struct arithmetic_sequence_data *d;

    d = malloc(sizeof(*d));
    if(d == 0) return 0;

    d->cur = x - a;  /* back up so first value returned is x */
    d->step = a;

    return create_sequence(arithmetic_sequence_next, d);
}

/* Return the sum of two sequences */
static int add_sequences_next(void *data) {
    Sequence *s;

    s = data;
    return sequence_next(s[0]) + sequence_next(s[1]);
}

Sequence add_sequences(Sequence s0, Sequence s1) {
    Sequence *s;

    s = malloc(2*sizeof(*s));
    if(s == 0) return 0;
```

```
    s[0] = s0;
    s[1] = s1;

    return create_sequence(add_sequences_next, s);
}

/* Return the sequence x, f(x), f(f(x)), ... */
struct iterated_function_sequence_data {
    int x;
    int (*f)(int);
}

static int iterated_function_sequence_next(void *data) {
    struct iterated_function_sequence_data *d;
    int retval;

    d = data;

    retval = d->x;
    d->x = d->f(d->x);

    return retval;
}

Sequence iterated_function_sequence(int (*f)(int), int x0) {
    struct iterated_function_sequence_data *d;

    d = malloc(sizeof(*d));
    if(d == 0) return 0;

    d->x = x0;
    d->f = f;

    return create_sequence(iterated_function_sequence_next, d);
}
```

Note that we haven't worried about how to free the data field inside a Sequence, and indeed it's not obvious that we can write a generic data-freeing routine since we don't know what structure it has. The solution is to add more function pointers to a Sequence, so that we can get the next value, get the sequence to destroy itself, etc. When we do so, we have gone beyond building a closure to building an object.

5. Objects

Here's an example of a hierarchy of counter objects. Each counter object has (at least) three operations: reset, next, and destroy. To call the next operation on counter c we include c and the first argument, e.g. c->next(c) (one could write a wrapper to enforce this).

The main trick is that we define a basic counter structure and then extend it to include additional data, using lots of pointer conversions to make everything work.

```
/* use preprocessor to avoid rewriting these */
#define COUNTER_FIELDS  \
    void (*reset)(struct counter *); \
    int (*next)(struct counter *);   \
    void (*destroy)(struct counter *);

struct counter {
    COUNTER_FIELDS
};

typedef struct counter *Counter;

/* minimal counter--always returns zero */
/* we don't even allocate this, just have one global one */
static void noop(Counter c) { ; }
static int return_zero(Counter c) { return 0; }
static struct counter Zero_counter = { noop, return_zero, noop };

Counter make_zero_counter(void) {
    return &Zero_counter;
}

/* a fancier counter that iterates a function sequence */
/* this struct is not exported anywhere */
struct ifs_counter {

    /* copied from struct counter declaration */
    COUNTER_FIELDS

    /* new fields */
    int init;
    int cur;
    int (*f)(int);    /* update rule */
};

static void ifs_reset(Counter c) {
```

```
    struct ifs_counter *ic;

    ic = (struct ifs_counter *) c;

    ic->cur = ic->init;
}

static void ifs_next(Counter c) {
    struct ifs_counter *ic;
    int ret;

    ic = (struct ifs_counter *) c;

    ret = ic->cur;
    ic->cur = ic->f(ic->cur);

    return ret;
}

Counter make_ifs_counter(int init, int (*f)(int)) {
    struct ifs_counter *ic;

    ic = malloc(sizeof(*ic));

    ic->reset = ifs_reset;
    ic->next = ifs_next;
    ic->destroy = (void (*)(struct counter *)) free;

    ic->init = init;
    ic->cur = init;
    ic->f = f;

    /* it's always a Counter on the outside */
    return (Counter) ic;
}
```

A typical use might be

```
static int times2(int x) {
    return x*2;
}

void print_powers_of_2(void) {
    int i;
    Counter c;

    c = make_ifs_counter(1, times2);

    for(i = 0; i < 10; i++) {
        printf("%d\n", c->next(c));
    }

    c->reset(c);

    for(i = 0; i < 20; i++) {
        printf("%d\n", c->next(c));
    }

    c->destroy(c);
}
```

[**1] https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)Iterators.html

C/Iterators

Note: You are looking at a static copy of the former PineWiki site, used for class notes by James Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions of some of these notes at http://www.cs.yale.edu/homes/aspnes/#classes.

1. The problem

Suppose we have an abstract data type that represents some sort of container, such as a list or dictionary. We'd like to be able to do something to every element of the container; say, count them up. How can we write operations on the abstract data type to allow this, without exposing the implementation?

To make the problem more concrete, let's suppose we have an abstract data type that represents the set of all non-negative numbers less than some fixed bound. The core of its interface might look like this:

1.1. nums.h

```
/*
 * Abstract data type representing the set of numbers from 0 to
 * bound-1 inclusive, where bound is passed in as an argument at creation.
 */
typedef struct nums *Nums;

/* Create a Nums object with given bound. */
Nums nums_create(int bound);

/* Destructor */
void nums_destroy(Nums);

/* Returns 1 if nums contains element, 0 otherwise */
int nums_contains(Nums nums, int element);
```

1.2. nums.c

```
#include <stdlib.h>
#include "nums.h"

struct nums {
    int bound;
};

Nums nums_create(int bound) {
    struct nums *n;
    n = malloc(sizeof(*n));
    n->bound = bound;
    return n;
}

void nums_destroy(Nums n) { free(n); }

int nums_contains(Nums n, int element) {
    return element >= 0 && element < n->bound;
}
```

From the outside, a Nums acts like the set of numbers from 0 to bound - 1; nums_contains will insist that it contains any int that is in this set and contains no int that is not in this set.

Let's suppose now that we want to loop over all elements of some Nums, say to add them together. In particular, we'd like to implement the following pseudocode, where nums is some Nums instance:

```
sum = 0;
for(each i in nums) {
    sum += i;
}
```

One way to do this would be to build the loop into some operation in nums.c, including its body. But we'd like to be able to substitute any body for the sum += i line. Since we can't see the inside of a Nums, we need to have some additional operation or operations on a Nums that lets us write the loop. How can we do this?

2. Option 1: Function that returns a sequence

A data-driven approach might be to add a nums_contents function that returns a sequence of all elements of some instance, perhaps in the form of an array or linked list. The advantage of this approach is that once you have the sequence, you don't need to worry about changes to (or destruction of) the original object. The disadvantage is that you have to deal with storage management issues, and have to pay the costs in time and space of allocating and filling in the sequence. This can be particularly onerous for a "virtual" container like Nums, since we could conceivably have a Nums instance with billions of elements.

Bearing these facts in mind, let's see what this approach might look like. We'll define a new function nums_contents that returns an array of ints, terminated by a -1 sentinel:

```
int *nums_contents(Nums n) {
    int *a;
    int i;
    a = malloc(sizeof(*a) * (n->bound + 1));
    for(i = 0; i < n->bound; i++) a[i] = i;
    a[n->bound] = -1;
    return a;
}
```

We might use it like this:

```
sum = 0;
contents = nums_contents(nums);
for(p = contents; *p != -1; p++) {
    sum += *p;
}
free(contents);
```

Despite the naturalness of the approach, returning a sequence in this case leads to the most code complexity of the options we will examine.

3. Option 2: Iterator with first/done/next operations
   If we don't want to look at all the elements at once, but just want to process them one at a time, we can build an iterator. An iterator is an object that allows you to step through the contents of another object, by providing convenient operations for getting the first element, testing when you are done, and getting the next element if you are not. In C, we try to design iterators to have operations that fit well in the top of a for loop.

   For the Nums type, we'll make each Nums its own iterator. The new operations are given here:

```
int nums_first(Nums n) { return 0; }
int nums_done(Nums n, int val) { return val >= n->bound; }
int nums_next(Nums n, int val) { return val+1; }
```

   And we use them like this:

```
sum = 0;
for(i = nums_first(nums); !nums_done(nums, i); i = nums_next(nums, i)) {
    sum += i;
}
```

   Not only do we completely avoid the overhead of building a sequence, we also get much cleaner code. It helps in this case that all we need to find the next value is the previous one; for a more complicated problem we might have to create and destroy a separate iterator object that holds the state of the loop. But for many tasks in C, the first/done/next idiom is a pretty good one.

4. Option 3: Iterator with function argument
   Suppose we have a very complicated iteration, say one that might require several nested loops or even a recursion to span all the elements. In this case it might be very difficult to provide first/done/next operations, because it would be hard to encode the state of the iteration so that we could easily pick up in the next operation where we previously left off. What we'd really like to do is to be able to plug arbitrary code into the innermost loop of our horrible iteration procedure, and do it in a way that is reasonably typesafe and doesn't violate our abstraction barrier. This is a job for function pointers, and an example of the functional programming style in action.

   We'll define a nums_foreach function that takes a function as an argument:

```
void nums_foreach(Nums n, void (*f)(int, void *), void *f_data) {
    int i;
    for(i = 0; i < n->bound; i++) f(i, f_data);
}
```

   The f_data argument is used to pass extra state into the passed-in function f; it's a void * because we want to let f work on any sort of extra state it likes.

   Now to do our summation, we first define an extra function sum_helper, which adds each element to an accumulator pointed to by f_data:

```
static void sum_helper(int i, void *f_data) {
    *((int *) f_data) += i;
}
```

   We then feed sum_helper to the nums_foreach function:

```
sum = 0;
nums_foreach(nums, sum_helper, (void *) &sum);
```

   There is a bit of a nuisance in having to define the auxiliary sum_helper function and in all the casts to and from void, but on the whole the complexity of this solution is not substantially greater than the first/done/next approach. Which you should do depends on whether it's harder to encapsulate the state of the iterator (in which case the functional approach is preferable) or of the loop body (in which case the first/done/next approach is preferable), and whether you need to bail out of the loop early (which would require special support from the foreach procedure, perhaps checking a return value from the function). However, it's almost always straightforward to encapsulate the state of a loop body; just build a struct containing all the variables that it uses, and pass a pointer to this struct as f_data.

5. Appendix: Complete code for Nums
   Here's a grand unified Nums implementation that provides all the interfaces we've discussed:

5.1. nums.h

```
/*
 * Abstract data type representing the set of numbers from 0 to
 * bound-1 inclusive, where bound is passed in as an argument at creation.
 */
typedef struct nums *Nums;

/* Create a Nums object with given bound. */
Nums nums_create(int bound);

/* Destructor */
void nums_destroy(Nums);

/* Returns 1 if nums contains element, 0 otherwise */
int nums_contains(Nums nums, int element);
```

```
/*
 * Returns a freshly-malloc'd array containing all elements of n,
 * followed by a sentinel value of -1.
 */
int *nums_contents(Nums n);

/* Three-part iterator */
int nums_first(Nums n);           /* returns smallest element in n */
int nums_done(Nums n, int val);   /* returns 1 if val is past end */
int nums_next(Nums n, int val);   /* returns next value after val */

/* Call f on every element of n with with extra argument f_data */
void nums_foreach(Nums n, void (*f)(int, void *f_data), void *f_data);
```

5.2. nums.c

```
#include <stdlib.h>
#include "nums.h"

struct nums {
    int bound;
};

Nums nums_create(int bound) {
    struct nums *n;
    n = malloc(sizeof(*n));
    n->bound = bound;
    return n;
}

void nums_destroy(Nums n) { free(n); }

int nums_contains(Nums n, int element) {
    return element >= 0 && element < n->bound;
}

int *nums_contents(Nums n) {
    int *a;
    int i;
    a = malloc(sizeof(*a) * (n->bound + 1));
    for(i = 0; i < n->bound; i++) a[i] = i;
    a[n->bound] = -1;
    return a;
}

int nums_first(Nums n) { return 0; }
int nums_done(Nums n, int val) { return val >= n->bound; }
int nums_next(Nums n, int val) { return val+1; }

void nums_foreach(Nums n, void (*f)(int, void *), void *f_data) {
    int i;
    for(i = 0; i < n->bound; i++) f(i, f_data);
}
```

And here's some test code to see if it all works:

5.3. test-nums.c

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

#include "nums.h"
#include "tester.h"

static void sum_helper(int i, void *f_data) {
    *((int *) f_data) += i;
}

int main(int argc, char **argv) {
    Nums nums;
    int sum;
    int *contents;
    int *p;
    int i;

    tester_init();

    TRY { nums = nums_create(100); } ENDTRY;
    TEST(nums_contains(nums, -1), 0);
    TEST(nums_contains(nums, 0), 1);
    TEST(nums_contains(nums, 1), 1);
    TEST(nums_contains(nums, 98), 1);
    TEST(nums_contains(nums, 99), 1);
```

```
    TEST(nums_contains(nums, 100), 0);

    sum = 0;
    contents = nums_contents(nums);

    for(p = contents; *p != -1; p++) {
        sum += *p;
    }

    free(contents);
    TEST(sum, 4950);

    sum = 0;

    for(i = nums_first(nums); !nums_done(nums, i); i = nums_next(nums, i)) {
        sum += i;
    }

    TEST(sum, 4950);

    sum = 0;
    nums_foreach(nums, sum_helper, (void *) &sum);
    TEST(sum, 4950);
    tester_report(stdout, argv[0]);
    return tester_result();
}

$> make test
gcc -g3 -ansi -pedantic -Wall -c -o test-nums.o test-nums.c
gcc -g3 -ansi -pedantic -Wall -c -o nums.o nums.c
gcc -g3 -ansi -pedantic -Wall -c -o tester.o tester.c
gcc -g3 -ansi -pedantic -Wall -o test-nums test-nums.o nums.o tester.o

$> ./test-nums
OK!


---
[**2]
https://www.cs.yale.edu/homes/aspnes/pinewiki/AbstractDataTypes.html

AbstractDataTypes

   Note: You are looking at a static copy of the former PineWiki site, used for class notes by James
   Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other
   bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions
   of some of these notes at http://www.cs.yale.edu/homes/aspnes/#classes.

Abstraction
   One of the hard parts about computer programming is that, in general, programs are bigger than
   brains. Unless you have an unusally capacious brain, it is unlikely that you will be able to
   understand even a modestly large program in its entirety. So in order to be able to write and debug
   large programs, it is important to be able to break it up into pieces, where each piece can be
   treated as a tool whose use and description is simpler (and therefor fits in your brain better) than
   its actual code. Then you can forget about what is happening inside that piece, and just treat it as
   an easily-understood black box from the outside.

   This process of wrapping functionality up in a box and forgetting about its internals is called
   abstraction, and it is the single most important concept in computer science. In these notes we will
   describe a particular kind of abstraction, the construction of abstract data types or ADTs. Abstract
   data types are data types whose implementation is not visible to their user; from the outside, all
   the user knows about an ADT is what operations can be performed on it and what those operations are
   supposed to do.

   ADTs have an outside and an inside. The outside is called the interface; it consists of the minimal
   set of type and function declarations needed to use the ADT. The inside is called the implementation;
   it consists of type and function definitions, and sometime auxiliary data or helper functions, that
   are not visible to users of the ADT.

Example of an abstract data type
   Too much abstraction at once can be hard to take, so let's look at a concrete example of an abstract
   data type. This ADT will represent an infinite sequence of ints. Each instance of the Sequence type
   supports a single operation seq_next that returns the next int in the sequence. We will also need to
   provide one or more constructor functions to generate new Sequences, and a destructor function to
   tear them down.

   Here is an example of a typical use of a Sequence:

void seq_print(Sequence s, int limit) {
    int i;

    for(i = seq_next(s); i < limit; i = seq_next(s)) {
        printf("%d\n", i);
    }
}
```

Note that seq_print doesn't need to know anything at all about what a Sequence is or how seq_next works in order to print out all the values in the sequence until it hits one greater than or equal to limit. This is a good thing--- it means that we can use with any implementation of Sequence we like, and we don't have to change it if Sequence or seq_next changes.

Interface

In C, the interface of an abstract data type will usually be declared in a header file, which is included both in the file that implements the ADT (so that the compiler can check that the declarations match up with the actual definitions in the implementation. Here's a header file for sequences:

sequence.h

```
/* opaque struct: hides actual components of struct sequence,
 * which are defined in sequence.c */
typedef struct sequence *Sequence;

/* constructors */
/* all our constructors return a null pointer on allocation failure */

/* returns a Sequence representing init, init+1, init+2, ... */
Sequence seq_create(int init);

/* returns a Sequence representing init, init+step, init+2*step, ... */
Sequence seq_create_step(int init, int step);

/* destructor */
/* destroys a Sequence, recovering all interally-allocated data */
void seq_destroy(Sequence);

/* accessor */
/* returns the first element in a sequence not previously returned */
int seq_next(Sequence);
```

Here we have defined two different constructors for Sequences, one of which gives slightly more control over the sequence than the other. If we were willing to put more work into the implementation, we could imagine building a very complicated Sequence type that supported a much wider variety of sequences (for example, sequences generated by functions or sequences read from files); but we'll try to keep things simple for now. We can always add more functionality later, since the users won't notice if the Sequence type changes internally.

Implementation

The implementation of an ADT in C is typically contained in one (or sometimes more than one) .c file. This file can be compiled and linked into any program that needs to use the ADT. Here is our implementation of Sequence:

sequence.c

```
#include <stdlib.h>
#include "sequence.h"

struct sequence {
    int next;  /* next value to return */
    int step;  /* how much to increment next by */
};

Sequence seq_create(int init) {
    return seq_create_step(init, 1);
}

Sequence seq_create_step(int init, int step) {
    Sequence s;

    s = malloc(sizeof(*s));
    if(s == 0) return 0;
    s->next = init;
    s->step = step;
    return s;
}

void seq_destroy(Sequence s) {
    free(s);
}

int seq_next(Sequence s) {
    int ret; /* saves the old value before we increment it */

    ret = s->next;
    s->next += s->step;

    return ret;
}
```

Things to note here: the definition of struct sequence appears only in this file; this means that only the functions defined here can (easily) access the next and step components. This protects Sequences to a limited extent from outside interference, and defends against users who might try to

"violate the abstraction boundary" by examining the components of a Sequence directly. It also means that if we change the components or meaning of the components in struct sequence, we only have to fix the functions defined in sequence.c.

Compiling and linking

Now that we have sequence.h and sequence.c, how do we use them? Let's suppose we have a simple main program:

main.c

```
#include <stdio.h>
#include "sequence.h"

void seq_print(Sequence s, int limit) {
    int i;

    for(i = seq_next(s); i < limit; i = seq_next(s)) {
        printf("%d\n", i);
    }
}

int main(int argc, char **argv) {
    Sequence s;
    Sequence s2;

    puts("Stepping by 1:");

    s = seq_create(0);
    seq_print(s, 5);
    seq_destroy(s);

    puts("Now stepping by 3:");

    s2 = seq_create_step(1, 3);
    seq_print(s2, 20);
    seq_destroy(s2);

    return 0;
 }
```

We can compile main.c and sequence.c together into a single binary with the command gcc main.c sequence.c. Or we can build a Makefile which will compile the two files separately and then link them. Using make may be more efficient, especially for large programs consisting of many components, since if we make any changes make will only recompile those files we have changed. So here is our Makefile:

Makefile

```
CC=gcc
CFLAGS=-g3 -ansi -pedantic -Wall

all: seqprinter

seqprinter: main.o sequence.o
    $(CC) $(CFLAGS) -o $@ $^

test: seqprinter
    ./seqprinter

# these rules say to rebuild main.o and sequence.o if sequence.h changes
main.o: main.c sequence.h
sequence.o: sequence.c sequence.h

clean:
    $(RM) -f seqprinter *.o
```

And now running make test produces this output. Notice how the built-in make variables $@ and $^ expand out to the left-hand side and right-hand side of the dependency line for building seqprinter.

```
$> make test
gcc -g3 -ansi -pedantic -Wall   -c -o main.o main.c
gcc -g3 -ansi -pedantic -Wall   -c -o sequence.o sequence.c
gcc -g3 -ansi -pedantic -Wall -o seqprinter main.o sequence.o

$> ./seqprinter
Stepping by 1:
0
1
2
3
4
Now stepping by 3:
1
4
7
10
13
```

16
19

Designing abstract data types

Now we've seen how to implement an abstract data type. How do we choose when to use when, and what operations to give it? Let's try answering the second question first.

Parnas's Principle

Parnas's Principle is a statement of the fundamental idea of information hiding, which says that abstraction boundaries should be as narrow as possible:

* The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide no other information.
* The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information.

(David Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," Communications of the ACM, 15(12): 1059--1062, 1972.)

For ADTs, this means we should provide as few functions for accessing and modifying the ADT as we can get away with. The Sequence type we defined early has a particularly narrow interface; the developer of Sequence (whoever is writing sequence.c) needs to know nothing about what its user wants except for the arguments passed in to seq_create or seq_create_step, and the user only needs to be able to call seq_next. More complicated ADTs might provide larger sets of operations, but in general we know that an ADT provides a successful abstraction when the operations are all "natural" ones given our high-level description. If we find ourselves writing a lot of extra operations to let users tinker with the guts of our implementation, that may be a sign that either we aren't taking our abstraction barrier seriously enough, or that we need to put the abstraction barrier in a different place.

When to build an abstract data type

The short answer: Whenever you can.

A better answer: The best heuristic I know for deciding what ADTs to include in a program is to write down a description of how your program is going to work. For each noun or noun phrase in the description, either identify a built-in data type to implement it or design an abstract data type.

For example: a grade database maintains a list of students, and for each student it keeps a list of grades. So here we might want data types to represent:

* A list of students,
* A student,
* A list of grades,
* A grade.

If grades are simple, we might be able to make them just be ints (or maybe doubles); to be on the safe side, we should probably create a Grade type with a typedef. The other types are likely to be more complicated. Each student might have in addition to his or her grades a long list of other attributes, such as a name, an email address, etc. By wrapping students up as abstract data types we can extend these attributes if we need to, or allow for very general implementations (say, by allowing a student to have an arbitrary list of keyword-attribute pairs). The two kinds of lists are likely to be examples of sequence types; we'll be seeing a lot of ways to implement these as the course progresses. If we want to perform the same kinds of operations on both lists, we might want to try to implement them as a single list data type, which then is specialized to hold either students or grades; this is not always easy to do in C, but we'll see examples of how to do this, too.

Whether or not this set of four types is the set we will finally use, writing it down gives us a place to start writing our program. We can start writing interface files for each of the data types, and then evolve their implementations and the main program in parallel, adjusting the interfaces as we find that we have provided too little (or too much) data for each component to do what it must.
