Building Dynamic ADTs

The C standard library doesn't contain any form of general purpose ADTs (sets, hash tables, lists, queues etc)

We can use dynamic memory to create our own ADTs that grow and shrink as required during execution

There are two basic approaches:

- One in which you allocate (and resize when necessary) a single chunk of memory - essentially a dynamic array - and use it to build your ADT
- The other in which you allocate many chunks of memory - usually pointers to structures - and link one to another in a linked list or tree like fashion

Why are there no containers in the C standard library?

ADTs in C tend to be specialised to each application, and therefore only contain the routines needed for that application

This tends to lead to **highly efficient code** - smaller binaries in particular - but **at the cost of some programmer time**

It is perfectly possible to write a "generic, one size fits all" implementation of some particular ADT, but it is more complex

Things to consider when designing your own container ADT modules:

- How much of the implementation does the container API expose?
- Are elements held by value or via pointers?
- Is the container responsible for performing its own dynamic memory allocation?
- How do you iterate over all the elements in a container?

A Minimal "growable string"

We often want to store a string buffer, into which we append several strings, but don't know what the upper limit of buffer length we need

The usual practice is to use an array of chars, and massively over-engineer the array size, documenting this clearly:

```
// assume we deal with strings typically 100-200 chars long

// deliberately over-engineer the upper limit:

#define MAXBUFSIZE 1024
typedef char mystrbuf[MAXBUFSIZE];
```

This is wasting space, and still doesn't completely fix the problem because we still need to check for buffer overrun - and then handle the error somehow

It would be better to build a buffer that starts small, but which can grow when needed

We can wrap a simple capacity and length shell around a single `malloc()`d chunk, and then use a pointer to that struct as our ADT:

```
struct dynstr {
        char *data; // single malloc()ed chunk
        int cap; // current allocated size of the chunk (the
capacity)
        int len; // how many elements stored in the chunk (len <
cap)
};

typedef struct dynstr *dynstr; // our ADT
```

We need a constructor, `dynstr_create()`, and a function to concatenate a new string onto the end of the existing string: `dynstr_strcat()`:

```
// dynstr ds = dynstr_create( s );
// Create and return a new dynstr, with initial data s

dynstr dynstr_create( char *s ) {
```

```
        dynstr new = malloc( sizeof(struct dynstr));
        assert( new != NULL );
        int len = strlen(s); int cap = 10+len;
        new->data = malloc(cap * sizeof(char));
        assert( new->data != NULL );
        strcpy( new->data, s );
        new->len = len;
        new->cap = cap;
        return new;
}
```

```
// dynstr_strcat( d, s );
// append string s onto the end of d's data

void dynstr_strcat( dynstr d, char *s ) {
        int len = strlen(s);
        d->len += len;
        if( d->len >= d->cap ) { // need more space?
                d->cap += 100+len;
                d->data = realloc( d->data, d->cap*sizeof(char) );
                assert( d->data != NULL );
        }
        strcat( d->data, s );
}
```

To round this off, we'll need a few more utility functions:

```
// int len = dynstr_len( d );
// Return the length of dynstr d

int dynstr_len( dynstr d ) {
        return d->len;
}

// char *data = dynstr_data( d );
// Return the dynstr's data; note: not a copy, the actual data

char *dynstr_data( dynstr d ) {
```

```
        return d->data;
}
```

Then finally, a free function `dynstr_free()`:

```
// void dynstr_free( d );
// When you've finished growing your dynstr, you should
// free the dynstr and it's data chunk

void dynstr_free( dynstr d ) {
        free( d->data );
        free( d );
}
```

In total, such a growable dynamic string requires you to write 70 lines in `dynstr.c` and another 20 lines in `dynstr.h`

A "growable generic dynamic array"

Let's see how we might generalise `dynstr` to be a generic dynamic array

When we say "generic" in C, we mean:

- Use `void *` pointers as our single element type
- Provide customised callbacks (function pointers) to print an element, free an element, and do any other generic operations (e.g. in some ADTs we might also need to duplicate an element, or compare elements - as we saw with `qsort()` and `bsearch()` earlier)
- Provide a generic `TYPE_print()` function to walk across every element in the collection, invoking the user-supplied *print element callback* for each element
- Provide a generic `TYPE_free()` function to free the collection. This walks across every element in the collection, invoking the user-supplied free element callback (if it's not NULL) for each element

Start by defining a macro `#define DA_ELEMENT void *` to make the dynarray element type easy to change, even though all of the code we're going to write will only currently work with `void *`'s

We do this in order to help if we ever clone and modify a specialised version of this code

- We define two function pointer types:

```
// "print an element" function pointer type
typedef void (*dynarray_printf)( FILE *out, DA_ELEMENT el, int pos );

// "free an element" function pointer type
typedef void (*dynarray_freef)( DA_ELEMENT el );
```

- The structure becomes:

```
struct dynarray {
  DA_ELEMENT *data;    // malloc()ed chunk of generic pointers
  int cap;             // current size of the chunk (the capacity)
  int len;             // #elements stored in the chunk (len <= cap)
  dynarray_freef fef;  // how to free an element
  dynarray_printf pef; // how to print an element
};
```

- Our constructor becomes:

```
// dynarray ds = dynarray_create( cap, pef, fef );
//   Create and return a new empty dynarray, with capacity cap,
//   print element function pef, and free element function fef
dynarray dynarray_create( int cap, dynarray_printf pef,
                          dynarray_freef fef ) {
  dynarray new = malloc( sizeof(struct dynarray));
  assert( new != NULL );
  new->data = malloc(cap * sizeof(DA_ELEMENT));
  assert( new->data != NULL );
  new->len = 0; new->cap = cap;
  new->fef = fef; new->pef = pef;
  return new;
}
```

- ... is dynarray_push(), shown along with it's grow() helper:

```
// Helper to grow the dynarray if it's full
static void grow( dynarray d ) {
  if( d->len+1 >= d->cap ) {    // need more space?
    d->cap += 100;
    d->data = realloc( d->data, d->cap*sizeof(DA_ELEMENT) );
    assert( d->data != NULL );
  }
}

// dynarray_push( d, element );
//   append element onto the end of d's data
void dynarray_push( dynarray d, DA_ELEMENT element ) {
  grow( d );
  d->data[ d->len++ ] = element;
}
```

- Our equivalent of dynstr_strcat(), that adds a single item...

- Our free operation is as follows (see how it uses the fef callback, unless it's NULL):

```
// dynarray_free( d );
//   free the dynarray elements (using the free element callback),
//   the data chunk, and the dynarray
void dynarray_free( dynarray d ) {
  if( d->fef != NULL ) {
    for( int i=0; i<d->len; i++ ) {
      (*d->fef)( d->data[i] );
    }
  }
  free( d->data );
  free( d );
}
```

- Let's add a generic print me out function:

```
// dynarray_print( d, out );
//   print function - print dynarray as comma-separated, []-delimited,
//   list of elements (using the print element callback to
//   print each element)
void dynarray_print( dynarray d, FILE *out ) {
  assert( d->pef != NULL );
  fputs( "[ ", out );
  for( int i=0; i<d->len; i++ ) {
    if( i>0 ) fputc( ',', out );
    (*d->pef)( out, d->data[i], i );
  }
  fputs( " ]", out );
}
```

- Finally, we'll also provide a generic dynarray_foreach() function to act as an iterator walking across every element in the collection, invoking a user-supplied process each element callback.
- That will require one more function pointer type:

```
// "process an element" function pointer type (for "foreach")
typedef void (*dynarray_foreachf)( DA_ELEMENT el, int pos, void *stateptr );
```

- The dynarray_foreach() function is (we'll see an example of how to call this shortly):

```
// dynarray_foreach( d, cb, stateptr );
//   iterator function: call cb( element, position, stateptr )
//   foreach element in d.
void dynarray_foreach( dynarray d, dynarray_foreachf cb, void *stateptr ) {
  assert( cb != NULL );
  for( int i=0; i<d->len; i++ ) {
    (*cb)( d->data[i], i, stateptr );
  }
}
```

- Note that there are a couple of other functions inherited from dynstr: dynarray_len() and dynarray_data(). The latter deliberately returns a void * to allow typecast-free assignment of the returned value to any pointer type.

- You'll find this example - together with a unit test program testdynarray.c - in example adts/dynarray_generic.
- testdynarray.c starts by defining a person type and operations on it:

```
#define NAMELEN 25
struct person {
  char firstname[NAMELEN];
  char lastname[NAMELEN];
  int age;
};
typedef struct person *person;
```

- Then some operations on person:

```
static person person_make( char *first, char *last, int age ) {
  person p = malloc( sizeof(struct person) );
  assert( p != NULL );
  COPYNAME( p->firstname, first );
  COPYNAME( p->lastname, last );
  p->age = age;
  return p;
}
```

```
static void person_print( FILE *out, DA_ELEMENT el, int pos ) {
  person p = el;
  fprintf( out, "%s %s (age %d)",
    p->firstname, p->lastname, p->age );
}
```

- Note that COPYNAME(dst,src) is a macro wrapped around strncpy() that makes sure buffer overrun doesn't occur, but that the name is NUL-terminated.

- testdynarray.c's main() constructs a dynarray of person, using the system free() function as the free element function:

```
dynarray d = dynarray_create( 10, &person_print, &free );
```

- Then it adds some familiar looking people:

```
dynarray_push( d, person_make("duncan", "white", 45) );
dynarray_push( d, person_make("will", "knottenbelt", 35) );
```

- Then it prints them out - which prints the text in the right hand column:

```
fputs( "prints as ", stdout );
dynarray_print( d, stdout );
putchar( '\n' );
```

```
prints as [ duncan white (age 45),will knottenbelt (age 35) ]
```

- Then, using the iterator, it sums up the ages of all the people in the array, using the address of a local variable int totalage - as the state pointer:

```
int totalage = 0;
dynarray_foreach( d, &psumagecb, &totalage );
testint( totalage, 80, "(dcw,will).totalage" );
```

```
static void psumagecb( DA_ELEMENT el, int pos, void *stateptr ) {
  person p = el; int *total = stateptr;
  *total += p->age;
}
```

- testdynarray.c's main() contains many other tests, and eventually frees the dynarray of people:

```
dynarray_free( d );
```

- Then testdynarray.c carries on - creating a second dynarray, this time a dynarray of string literals, supplying an entirely different print element function and a NULL free element function -because string literals don't need free()ing:

```
static void str_print( FILE *out, DA_ELEMENT el, int pos ) {
  fputs( (char *)el, out );
}

// now create a dynarray of string literals..
d = dynarray_create( 10, &str_print, NULL );
```

- testdynarray.c carries on - adding strings, testing various properties, printing the dynarray out, and finally frees the second dynarray and terminates.
- Examine testdynarray.c carefully, then compile and run it with valgrind: not only do all tests pass, but no memory is leaked.

Our minimal `dynarray` module, like the earlier `dynstr` module, only allows appending of elements onto the end of a `dynarray`

By calling the appending process `pushing`, perhaps we suggested a slightly stack-like approach. In example `adts/dynarray_extended`, we add a `pop()` operator to make it even more stack-like

With only a bit more thought, we can add a `dequeue()` operator to remove an item off the front of the array - shuffling the rest of the items down one place - and now our `dynarray` can also act as a queue by enqueuing items with `dynarray_push()` and dequeuing them with `dynarray_dequeue()`

To do this we need two functions and a macro

Macro:
`#define dynarray_enqueue(d,p) dynarray_push(d,p)`

to allow us to enqueue by calling `dynarray_enqueue()` rather than `dynarray_push()` even though the code is the same

```
// DA_ELEMENT element = dynarray_pop( d );
// Pop the last element off the end of d, and return it.
// Precondition: d must not be empty.
// The caller must free the element.
DA_ELEMENT dynarray_pop( dynarray d ) {
        assert( d->len > 0 ); // not empty
        DA_ELEMENT result = d->data[ --d->len ];
        d->data[ d->len ] = NULL;
        return result;
}
```

```
// DA_ELEMENT element = dynarray_dequeue( d );
// Dequeue the first element off the front of d, and return it.
// Precondition: d must not be empty.
// The caller must free the element.
DA_ELEMENT dynarray_dequeue( dynarray d ) {
        assert( d->len > 0 ); // not empty
        DA_ELEMENT result = d->data[ 0 ];
        for( int i=1; i<d->len; i++ ) {
                d->data[i-1] = d->data[i];
        }
        d->data[ --d->len ] = NULL;
        return result;
}
```

`dequeue()` is much less efficient than the other operations, due to it shifting all the elements down one, an `O(n)` operation

There are ways round that - in particular, array-based queue implementations often use the array as a circular buffer, in which the front of the queue doesn't have to be element 0

That would be easy enough to implement, but would probably require splitting such a queue module off from dynarray completely, and implementing enqueue() and other routines differently. Let’s not do that here

Example Test code:

```
printf( "enqueue duncan\n" );
dynarray_enqueue( d, person_make("duncan", "white", 45) );
fputs( "(d): prints as: ", stdout );
dynarray_print( d, stdout ); putchar( '\n' );
printf( "enqueue will\n" );
dynarray_enqueue( d, person_make("will", "knottenbelt", 35) );
fputs( "(d,w): prints as: ", stdout );
dynarray_print( d, stdout ); putchar( '\n' );
printf( "deq: should get duncan.. let's check\n" );
person who = dynarray_dequeue( d );
printf( "deq: dequeued person is " );
person_print( stdout, who, 0 ); putchar( '\n' );
free( who );
fputs( "(d,w,-d) should print as w: ", stdout );
dynarray_print( d, stdout ); putchar( '\n' );
```

When run, this produces:

```
enqueue duncan
(d): prints as: [ duncan white (age 45) ]
enqueue will
(d,w): prints as:
        [ duncan white (age 45), will knottenbelt (age 35) ]
deq: should get duncan.. let's check
deq: dequeued person is duncan white (age 45)
(d,w,-d) should print as w: [ will knottenbelt (age 35) ]
```

Slightly reformatted for width

It’s arguable that instead of adding stack functions to a dynarray, it would be better to separate out a dynarray-based stack module, and similarly a dynarray acting as

a queue might be better separated out

If you decide to do this, rather than trying to implement some form of subclassing in C, there's nothing wrong with either:

- Building a stack or queue module as a thin layer on top of a dynarray, making the stack/queue module ADT a type synonym for a dynarray, or
- Cloning the dynarray module as a stack (or queue) module, i.e. physically copying the files and then modifying the copy

The first option may be less work

It requires the dynarray struct type to be visible in dynarray.h so that the stack module's extra stack pop() operator can manipulate the internals of the dynarray struct correctly

The second option (clone and modify) breaks the DRY principle. If you derive a queue from a dynarray, then later discover and fix a bug in dynarray, you should fix a corresponding bug in the queue too

However, it allows deeper customisation of each module, e.g. removing inappropriate dynarray operations from a queue – or changing the underlying data type

Note that a dynarray can support insertion in the middle relatively easily, all you have to do, having picked your insertion index, is to shuffle higher elements up one to make space, and then add the new element

Similarly, you can easily enough write a delete element N operator if you need to be able to delete arbitrary elements from your dynarray

Just make sure you are clear whether deleting an element from a dynarray frees the element, or delivers the element (along with the responsibility to free the element later) to the caller

One typical use of insert in the middle would be to provide sorted dynamic arrays that insert elements in the right place via an element comparison function

How might we do this? First, add a pointer to a `qsort()` / `bsearch()` compatible comparator function to the structure:

```
// "compare two elements" function pointer type
// nb: a comparison function must return a -ve number if a<b,
```

```
// 0 if a==b or a +ve number if a>b
typedef int (*dynarray_comparef)( DA_ELEMENT a, DA_ELEMENT b );
struct dynarray {
        ....
        dynarray_comparef cf; // how to compare two elements
};
```

Then add a way of setting it, either in the constructor or as a separate setter (having initialised the field to NULL in the constructor):
`void dynarray_setcomparator( dynarray d, dynarray_comparef cf ) { d->cf = cf; }`

Finally, add our new `addsorted` operator:

```
// dynarray_addsorted( d, element );
// add element into d's data at the right sorted position
// pre: d's comparison function cf must have been set first
void dynarray_addsorted( dynarray d, DA_ELEMENT element ) {
        assert( d->cf != NULL );
        grow( d );
        int ipos; // the insertion point: the 1st pos whose item >=
element
        for( ipos=0; ipos < d->len && (*d->cf)( d->data[ipos],
element ) < 0; ipos++ )
                /*EMPTY*/;
        printf( "debug: addsorted: insertion point is %d\n", ipos
);
        for( int i=d->len-1; i>=ipos; i-- ) {
                d->data[i+1] = d->data[i];
        }
        d->len++;
        d->data[ipos] = element;
}
```

The final thing to mention about the *use `void *` and function pointers** approach to generics that we've shown in these dynamic arrays, is that it works for all pointer-based elements

But it doesn't help if you want a dynamic array of a builtin type such as a double

In such a case, I see no alternative to using clone and modify to make a specialised version `doubledynarray.[ch]` with the `DA_ELEMENT macro` renamed as `DDA_ELEMENT` and changed to double, and all public functions' names being prefixed with double, the generic print element function replaced by simple inline code, and the generic free element function removed altogether

Look in example `adts/dynarray_double`

Singly Linked Lists - intlists

This brings us to the second basic approach we mentioned above, in which you allocate many chunks of memory - usually pointers to structures - and link one to another in a linked list or tree like fashion

Start with the singly linked list

Suppose we want a "list of integers" ADT, which we'll call an `intlist`, which we'd like to model on the Haskell inductive data type: `data intlist = nil | cons int intlist`

Let's build the `intlist` module from first principles as a singly linked list

Singly Linked Lists - `intlist`s - version 1

Let's start by seeing how we'd like to use our hypothetical intlists

Let's make our first tiny example and have it be the equivalent of Haskell's `cons(300,cons(200,cons(100,nil)))`:

```
#include <stdio.h>
#include <stdlib.h>
#include "intlist.h"
int main( void ) {
        intlist l = intlist_cons( 300, intlist_cons( 200,
intlist_cons( 100, intlist_nil() ) ) );
        return(0);
}
```

We write - without thinking - `intlist l` to declare an `intlist` variable. This is a further example of why hiding the '`*`' is a good idea, in this case hiding the '`*`' inside the `intlist` type definition

The core of our `intlist` ADT is the data structure itself. Let's store a "nil-shaped" list as the NULL pointer, and store a non-nil list (each "cons-shaped" node) as a pointer to a structure which stores the head (an integer) and the tail (a pointer to the rest of the list):

```
struct intlist;
typedef struct intlist *intlist;
struct intlist {
        int head;
        intlist tail;
};
```

Note that we're calling the structure `struct intlist` and also calling the pointer to it `intlist`

This may seem even more self-referential than usual here, because `struct intlist` contains an `intlist` (pointer) within it

But this is the idiomatic way

Ok, we can now write our two Haskell-style constructors:

```
#define NEW(t) malloc(sizeof(struct t))
intlist intlist_nil( void ) { return NULL; }
intlist intlist_cons( int head, intlist tail ) {
        intlist new = NEW(intlist);
        assert( new != NULL );
        new->head = head;
        new->tail = tail;
        return new;
}
```

In fact, `intlist_nil()` is so simple it hardly needs to be a real function, we might as well make it a macro in `intlist.h`: `#define intlist_nil() NULL`

We have now built enough of our intlist infrastructure to compile and run `listeg1` from a couple of slides ago

When you compile this, because we don't yet do anything with our `intlist`, `gcc` gives a warning

Of course, if we run our `listeg1` executable, nothing appears to happen - because we haven't yet written any code to print out the list!

But we can use `gdb` to see our list

We can either single step through the whole program as it builds our 3 element list, or set a breakpoint once it's built the whole list – before `main()` returns

- Here I do the latter: The return 0 at the end of main() is line 18. So I run:

```
gdb listeg1
(gdb) b 18              [set a breakpoint at line 18]
(gdb) r                 [run the program until it hits the breakpoint]
Breakpoint 1, main () at listeg1.c:18
18   return 0;
```

- Now, to examine our intlist l, we can:

```
(gdb) p l
$1 = (intlist) 0x5555555592e0
(gdb) p *l
$2 = {head = 300, tail = 0x5555555592c0}
(gdb) p *(l->tail)
$3 = {head = 200, tail = 0x5555555592a0}
(gdb) p *(l->tail->tail)
$4 = {head = 100, tail = 0x0}
```

Here we see the "first head" is 300, the second 200, the third 100, and the final tail is "0x0" - NULL (the exact pointer addresses you see will be different)

Singly Linked Lists - `intlist`s - version 2

The next thing to do is to add the ability to print out the members of the list

Let’s extend our tiny example, making it print out the list as it builds it:

```
#include <stdio.h>
#include <stdlib.h>
#include "intlist.h"
int main( void ) {
        intlist l = intlist_nil(); printf( "the list is " );
print_intlist( stdout, l ); putchar( '\n' );
        l = intlist_cons( 100, l ); printf( "the list is " );
print_intlist( stdout, l ); putchar( '\n' );
        l = intlist_cons( 200, l ); printf( "the list is " );
print_intlist( stdout, l ); putchar( '\n' );
        l = intlist_cons( 300, l ); printf( "the list is " );
print_intlist( stdout, l ); putchar( '\n' );
        return 0;
}
```

When run, we’d like it to report something like:

```
the list is [ ]
the list is [ 100 ]
the list is [ 200,100 ]
the list is [ 300,200,100 ]
```

How do we implement `print_intlist()`:

```
void print_intlist( FILE *f, intlist l ) {
        fputs( "[ ", f );
        while( l != NULL ) {
                fprintf( f, "%d", l->head );
                l = l->tail;
                if( l != NULL ) putc( ',', f );
        }
```

```
        fputs( " ]", f );
}
```

The core of that is:

```
while( l != NULL ) {
        // deal with l->head
        l = l->tail;
}
```

Or, with even greater clarity, as:

```
for( ; l != NULL; l = l->tail ) {
        // deal with l->head
}
```

Both forms are highly idiomatic and well worth memorising

Compiling and running generates our desired output

The next thing to add is a mechanism equivalent to Haskell's pattern matching, i.e. a general method that allows us to write list-processing functions

Note that we don't want our clients to use knowledge of the underlying representation (that NULL represents `nil()`, that the head of a non-nil list can be found in `l->head` etc)

This information should remain **private to intlist.c**, in case we want to change it later

Haskell-style pattern matching does two things:

- Tell us which shape a given list is - is it nil or non-nil (cons)?
- When we know that we have a `cons(h,t)` node, break it apart into `h` and `t`

Looking at the first of those, as there are only two such 'shapes' (nil and cons) the obvious way to tell us which shape a list is is to write the following function, which returns true if `l` is nil, false if `l` is `cons(h,t)`:

```
bool intlist_is_nil( intlist l ) {
        return l == NULL;
}
```

Let's extend our tiny example of using lists, first we write a function to describe whether the list is nil or cons:

```
static void describe( intlist l ) {
        printf( "the list is " ); print_intlist( stdout, l );
        if( intlist_is_nil(l) ) {
                printf( ": nil\n" );
        } else {
                printf( ": cons\n" );
        }
}
```

Then calling `describe()` as we build and print our list:

```
int main( void ) {
        intlist l = intlist_nil();
        describe( l );
        l = intlist_cons( 100, l );
        describe( l );
        ...
        return 0;
}
```

Compile this and run it, and you'll see that it reports:

```
the list is [ ]: nil
the list is [ 100 ]: cons
the list is [ 200,100 ]: cons
the list is [ 300,200,100 ]: cons
```

So this enables us to find out which 'shape' of `intlist` we have

The second thing that pattern matching does is *When we know that we have a `cons(h,t)` node, break it apart into `h` and `t`*

We can do this by writing the following function:

```
// int h; intlist t;
// get_intlist_cons( l, &h, &t );
// Given a non-nil (cons) list, extract the head and tail
// from that cons node into the h and t variables
void get_intlist_cons( intlist l, int *hp, intlist *tp ) {
        assert( l != NULL );
        *hp = l->head;
        *tp = l->tail;
}
```

As the comment suggests, this is intended to be called like:

```
if( ! intlist_is_nil(l) ) {
        int h; intlist t;
        get_intlist_cons( l, &h, &t );
        // now deal with h and t
}
```

here we are using the usual simulate var parameters by passing pointers trick to allow `get_intlist_cons()` to write results into `h` and `t`

Let's extend our `describe()` helper function a little:

```
static void describe( intlist l ) {
        printf( "the list is " ); print_intlist( stdout, l );
        if( intlist_is_nil(l) ) {
                printf( ": nil\n" );
        } else {
                int h; intlist t;
                get_intlist_cons( l, &h, &t );
                printf( ": cons, head = %d, tail = %p\n", h, t );
        }
}
```

Here, in the else part (when the list is cons) we break `l` apart into it's head (`h`) and tail (`t`), and then print out both - using `printf()`'s little known `%p` print

specifier to print an arbitrary pointer

Compile this and run it, and you'll see that it reports:

```
the list is [ ]: nil
the list is [ 100 ]: cons, head = 100, tail = (nil)
the list is [ 200,100 ]: cons, head = 200, tail = 0x5652494de6b0
the list is [ 300,200,100 ]: cons, head = 300, tail =
0x5652494de6d0
```

We now have enough infrastructure to translate small Haskell list functions into C code

Let's write client-side functions equivalent to the Haskell-ish:

```
length nil = 0
length cons(h,t) = 1 + length t
sumup nil = 0
sumup cons(h,t) = h + sumup t
```

For the `length()` example, we can do this by writing the following function:

```
static int length( intlist l ) {
        int result = 0;
        while( ! intlist_is_nil(l) ) {
                int h;
                get_intlist_cons( l, &h, &l );
                result++;
        }
        return result;
}
```

Note that here we write the tail of `l` back directly into `l`, by passing `&l` into `get_intlist_cons()` as it's `lp` parameter

This looks weird, but is perfectly safe

The `sumup()` function is very similar:

```
static int sumup( intlist l ) {
        int result = 0;
        while( ! intlist_is_nil(l) ) {
                int h;
                get_intlist_cons( l, &h, &l );
                result += h;
        }
        return result;
}
```

Let's call both functions from `describe()`, replacing the `nil` or `cons` information:

```
static void describe( intlist l ) {
        printf( "the list is " ); print_intlist( stdout, l );
        int len = length( l );
        int sum = sumup( l );
        printf( ", len=%d, sum=%d\n", len, sum );
}
```

Compile this and run it, and you'll see that it reports:

```
the list is [ ], len=0, sum=0
the list is [ 100 ], len=1, sum=100
the list is [ 200,100 ], len=2, sum=300
the list is [ 300,200,100 ], len=3, sum=600
```

You may wonder why we need to write two similar functions as `length()` and `sumup()`

Aren't we repeating ourselves and also doing two passes over the data? Yes we are

There are several ways of avoiding this. Let's focus on one such way: let's assume that we are always going to want to calculate both the length and the sum

So we can merge them together, calculating both results in a single pass and using simulated var parameters to return multiple results as usual:

```
static void len_sum( intlist l, int *lp, int *sp ) {
        *lp = 0; *sp = 0;
        while( ! intlist_is_nil(l) ) {
                int h;
                get_intlist_cons( l, &h, &l );
                (*lp)++; (*sp) += h;
        }
}
```

We now call this from our modified `describe()` function:

```
static void describe( intlist l ) {
        printf( "the list is " ); print_intlist( stdout, l );
        int len, sum;
        len_sum( l, &len, &sum );
        printf( ", len=%d, sum=%d\n", len, sum );
}
```

Here, we have effectively built a client-side iterator using our low-level deconstruction functions `intlist_is_nil()` and `get_intlist_cons()`

But in our `dynarray` module we provided a more general iterator called `dynarray_foreach()` which iterated over the `dynarray` and called a per-element callback

This allowed us to concentrate on the operation rather on the mechanics of traversing the list. Let's do something like that for intlists:

First, in `intlist.h` we add a `for-each-element` callback type:

```
// A foreach callback function, taking the head and a generic state
pointer
typedef void (*intlist_foreachf)( int, void * );
```

Second, in `intlist.c` we add:

```
// foreach_intlist( l, &mycbfunc, &mystate );
// Intlist iterator: iterate over l, invoking mycbfunc()
```

```
// once per head element, with the head and the state pointer,
// usually a pointer to some kind of struct.
void foreach_intlist( intlist l, intlist_foreachf cb, void *state )
{
        for( ; l != NULL; l = l->tail ) {
                (*cb)( l->head, state );
        }
}
```

Note that we called our iterator `foreach_intlist`, whereas it's dynarray equivalent was called `dynarray_foreach`. While consistency is good, we don't have to be slavishly consistent.

I quietly reserved the pattern `intlist_C()` for constructor functions, and used different patterns for other types of functions, e.g. `intlist_is_C()` for shape testers, `get_intlist_C()` for shape deconstructors, and `X_intlist()` for all other functions - including `foreach_intlist()`

Anyway, how might we use `foreach_intlist()` to calculate lengths and sums in a single pass?

Look in `listeg6.c`: **the caller and the callback must agree on what the generic state pointer is** - here we define a structure type with `len` and `sum` fields:
`typedef struct { int len, sum; } ls_state;`

Second, we define a suitable callback function:

```
static void len_sum( int head, void *state ) {
        ls_state *sp = state;
        sp->len++; sp->sum += head;
}
```

Third, in `describe()` we call `foreach_intlist()` this way:

```
ls_state ls = { 0, 0 };
foreach_intlist( l, &len_sum, &ls );
printf( ", len=%d, sum=%d\n", ls.len, ls.sum );
```

For our 7th version of `intlist`, we'll add three important functions: `free_intlist()` to free our list, an optimised `len_intlist()` to calculate lengths without needing to invoke an iterator, and `sprint_intlist()` (which is like `print_intlist()` but which prints into a string buffer)

The optimized `len_intlist()` operator:

```
int len_intlist( intlist l ) {
        int len = 0; for( ; l != NULL; l = l->tail ) len++; return
len;
}
```

`sprint_intlist()` replaces `printf()` calls with `sprintf()` and `strcat()` calls, assuming that the string buffer is big enough:

```
// sprint_intlist( buf, l );
// "string print" list l to char buf[] as []-wrapped csv data
//
void sprint_intlist( char *s, intlist l ) {
        strcpy( s, "[ " );
        while( l != NULL ) {
                char tempstr[20];
                sprintf( tempstr, "%d", l->head ); strcat( s,
tempstr );
                l = l->tail; if( l != NULL ) strcat( s, "," );
        }
        strcat( s, " ]" );
}
```

`free_intlist()`:

```
void free_intlist( intlist l ) {
        while( l != NULL ) {
                intlist tail = l->tail; free( l ); l = tail;
        }
}
```

We modify `describe()` to call `len_intlist()` instead of `foreach_intlist()`, and call `free_intlist(l)`; at the end of `main()`, If you compile it and run it, it behaves as expected

If you run `valgrind ./listeg7` it reports that all memory blocks have been `free()`d

We'll add an important capability that our original `dynarray` had - the `intlist_addsorted(v,l)` operator which gives us the ability to insert the element `v` into an (already sorted) list `l` in order to construct a bigger still sorted list

We saw the basic algorithm needed back in the `dynarray` code, all we need do is translate it into the linked list context

However, there are more special cases - the list can be NULL, or the insertion point can be in front of the first node, between any two nodes, or after the last node

Our `intlist_addsorted( v, l )` function starts as follows:

```
// l = intlist_addsorted( v, l );
// Given a value v and a sorted intlist l (nil or not)
// add v to the list at the right place so that the list REMAINS
SORTED;
// returning the modified list.
//
intlist intlist_addsorted( int v, intlist l ) {
        intlist new = intlist_cons( v, NULL ); // a new node with
head v.
        if( l == NULL ) return new; // if l is nil, return new node
        // must insert our new node at the right place:
        if( l->head > v ) { // add new node right at the front
                new->tail = l;
                return new;
        }
}
```

In all other cases, we must insert our new node after one or more existing nodes. So we must find which node to insert our new node after - call this the before insertion point, *bip*.
bip can be the very last node - `bip->head <= v && bip->tail == NULL` (if the new value is bigger than all values on the list) - or some intermediate node where
`bip->head <= v && bip->tail != NULL && bip->tail->head > v`.
We find this by:

```
intlist bip;
for( bip=l;
         bip->tail!=NULL && bip->tail->head<=v;
         bip=bip->tail ) /*EMPTY*/;
assert( bip != NULL );
assert( bip->head <= v );
assert( bip->tail == NULL || bip->tail->head > v );
#ifdef BIP_DEBUG
printf( "debug: addsorted(v=%d): bip is %p, head %d, tail %p\n",
  v, bip, bip->head, bip->tail );
#endif
```

Notice the asserts to enforce the loop post-condition

To complete `intlist_addsorted()` we attach the new node after `bip`, making sure that whatever originally followed `bip` now follows the new node:

```
        new->tail = bip->tail;
f       bip->tail = new;
        return l;
}
```

Finally, to make our main program call `intlist_addsorted` instead of `intlist_cons`, we either have to globally replace cons with `addsorted`, or alternatively we can use the C pre-processor to do the job for us

To do the latter, we add the following above `main()`:
`#define intlist_cons(a,b) intlist_addsorted(a,b)`

We also change the sequence of elements we add, in order to better exercise inserting elements into a sorted list

We've now developed the whole `intlist` module

Generic Singly Linked Lists

How can we generalise `intlist` - suppose we wanted a list of people, not integers

We can take `intlist.[ch]` and turn it into a generic linked list ADT, i.e. a singly linked list where the head is a generic `(void *)` pointer

See example `adts/linkedlist_generic` which does precisely this

Most of the techniques are precisely the same as we used in our generic dynarray - void * pointers and function callbacks to perform the operations that vary depending on the real type of the elements

But in this case we do have the problem of where to store the function callbacks, and require (gasp) two separate pointer types for almost the first time

Doubly Linked Lists of Integers

Next, let's look at the implementation of a doubly linked list - specifically a list of integers

Each value in the list will be held in a different list node, or `dllink`, which also holds link pointers to the previous and following elements

These pointers will allow us to move backwards and forwards through the list

In addition, in the whole list, we'll store a pointer to the front element and a separate pointer to the back element - to make it easy to start traversing the list from either end. We need two pointer-to-struct types:

- A `dllist` represents the whole list, holding the front and back link pointers:

```
typedef struct dllist *dllist;
struct dllist {
  dllink front;   // pointer to first element of list
  dllink back;    // pointer to the last element of list
};
```

- A `dllink` holds an element of the list, plus the next and previous pointers:

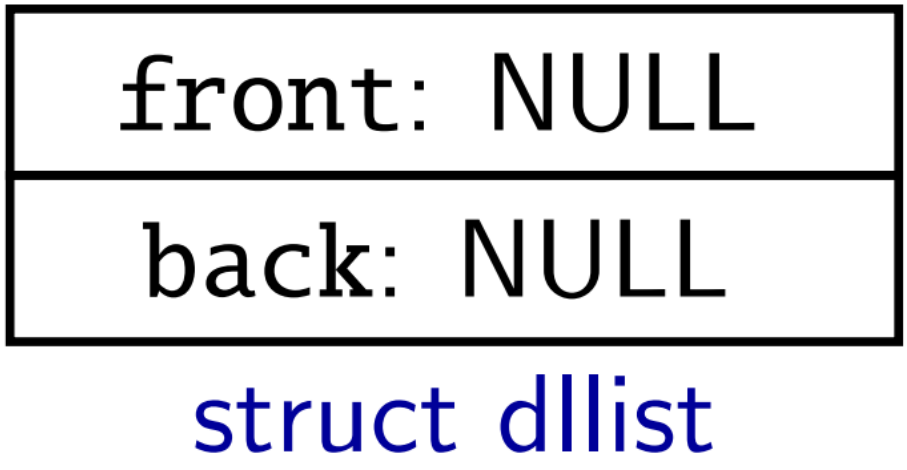
```
typedef struct dllink *dllink;
struct dllink {
  int value;     // element value
  dllink next;   // link to next element
  dllink prev;   // link to previous element
};
```

Let's look at snippets of code to create some doubly-linked lists, and see diagrams of what they do:

First, we create an empty doubly-linked list by:

```
dllist list = NEW(dllist);
assert( list != NULL );
list->front = NULL;
list->back = NULL;
```

Which builds:

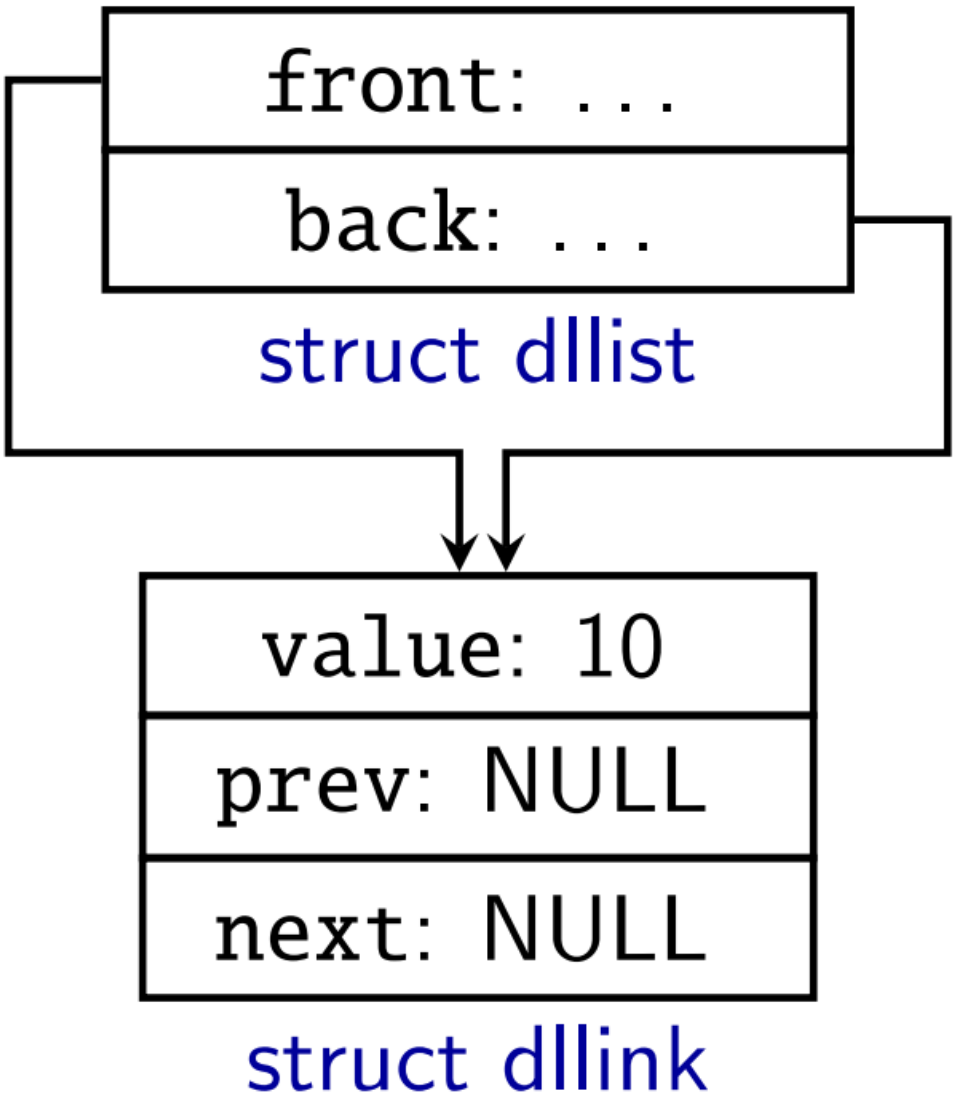


Next, we add a first element, with value 10 (let's say), to our doubly-linked list, modifying that list:

```
dllink new = NEW(dllink); // create new element with value 10
assert( new != NULL );
```

```
new->value = 10;
new->next = NULL;
new->prev = NULL;
if( list->front == NULL ) { // if the list was empty
        list->front = new; // new is front of list
        list->back = new; // and the back of the list..
}
```

- Which builds:

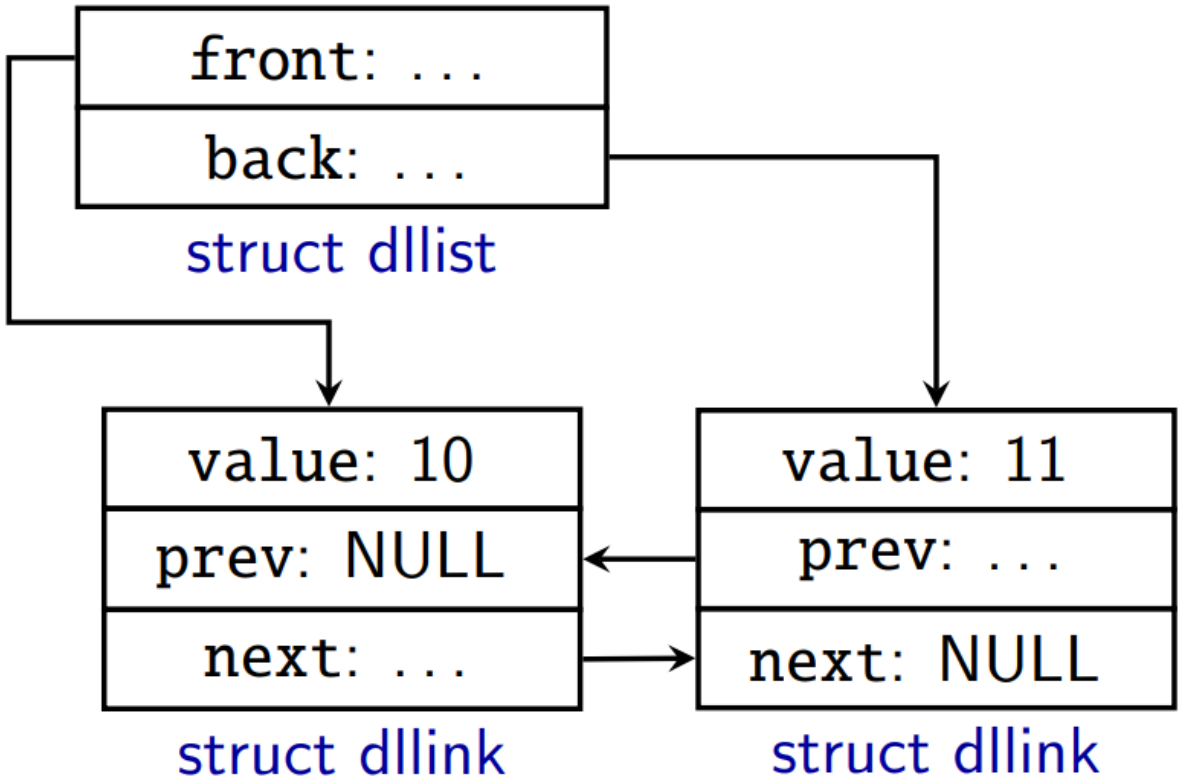


Next, we add a second element, with value 11 this time, to the end of our doubly-linked list, modifying it:

```
dllink new = NEW(dllink); // create new element with value 11
assert( new != NULL );
new->value = 11;
```

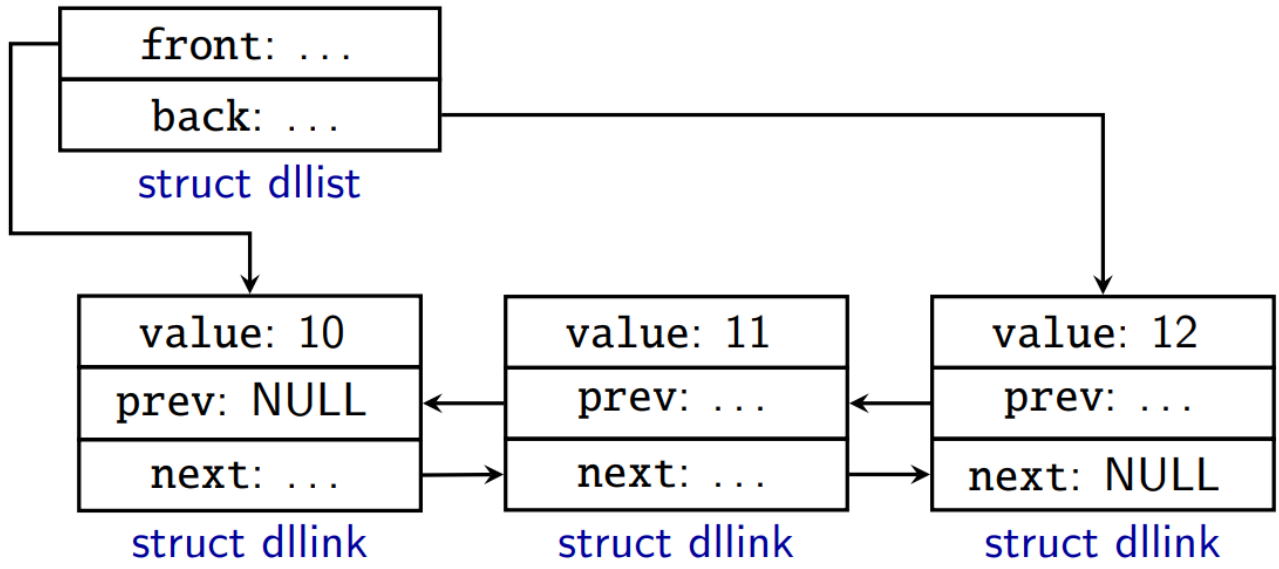
```
new->next = NULL;
new->prev = NULL;

if( list->front == NULL ) { // if the list was empty
        ...
} else { // otherwise..
        dllink last = list->back; // find the end of the list
        last->next = new; // link next(old-last) = new.
        new->prev = last; // backlink prev(new) = old-last
        list->back = new; // new is now the back of the list..
}
```

- Which builds:



Next, we add a third element, with value 12 this time, to the end of our doubly-linked list, modifying it:

The code is unchanged. The structure we end up with is:



Bringing the code together

Before we structure our doubly-linked list code as an ADT module, let's see it all together in one source file: `allinone.c`

After all the type definitions, and the usual `NEW()` macro, we find a function that we may use as follows:

```
dllist l = dl_build_list( from, to );
```

If executed with `from=10` and `to=12`, this would build a doubly-linked list containing the 3 integers 10..12, as shown in the previous diagram

- The body of dl_build_list() is as follows:

```
dllist list = NEW(dllist);     // first, create an empty list
assert( list != NULL );
list->front = NULL;
list->back = NULL;

for( int i=from; i <= to; i++ ) {
  dllink new = NEW(dllink);   // create new element
  assert( new != NULL );
  new->value = i;
  new->next = NULL;
  new->prev = NULL;
  printf("debug: make new node with value %d\n", new->value);

  // now link new onto the end of the list
  if( list->front == NULL ) { // if the list is initially empty
    list->front = new;        // new is front of list
    list->back = new;         // and the back of the list..
  } else {                    // otherwise..
    dllink last = list->back; // find the end of the list
    last->next = new;         // link next(old-last) = new.
    new->prev = last;         // backlink prev(new) = old-last
    list->back = new;         // new is now the back of the list..
  }
}
return list;
```

- You will recognise most of it as the assembled snippets from the preceding slides.

Refactorings

We notice that both the then-part and the else-part of the main if end with the same assignment `list->back = new`, we should move that statement below the entire if statement

We set `new->prev` twice: once when creating the element, setting `new->prev = NULL`, and a second time in the else part, setting `new->prev = last` (the old last, or back, element). So I moved the `new->prev = NULL` into the then part

Now, the then-part and else-part assign something to `new->prev` as the last thing both parts do.

- Wondering whether I could merge the assignments and place a single assignment to `new->prev` below the entire if, I realised that `last` is NULL if the list was initially empty. So we could lift our `last` variable up above the if, and write:

```
// now link new onto the end of the list
dllink last = list->back;   // find the end of the list
if( list->front == NULL ) { // if the list is initially empty
  list->front = new;        // new is front of list
} else {                    // otherwise..
  last->next = new;         // link next(old-last) = new.
}
new->prev = last;           // backlink prev(new) = old-last
list->back = new;           // new is now the back of the list..
```

- Next, I realised that we could test `list->back == NULL` instead of testing `list->front == NULL`, and that because `list->back` is currently stored in `last`, we could write that as `last == NULL`. So that became allinone3.c:

```
dllink last = list->back;   // find the end of the list
if( last == NULL ) {        // if the list is initially empty
  list->front = new;        // new is front of list
} else {                    // otherwise..
  last->next = new;         // link next(old-last) = new.
}
new->prev = last;           // backlink prev(new) = old-last
list->back = new;           // new is now the back of the list..
```

- Looking now at:

```
dllink last = list->back;   // find the end of the list
if( last == NULL ) {        // if the list is initially empty
  list->front = new;        // new is front of list
} else {                    // otherwise..
  last->next = new;         // link next(old-last) = new.
}
```

  we realise that we always perform one assignment into a dllink, and the value we are assigning is always `new`, but that the only variable is: which dllink we store it in.
- We might wonder whether you can assign directly to a `?:` expression:

But this doesn't compile!

```
(last == NULL ? list->front : last->next ) = new;
```

  (you'll find this bad version in allinone4-broken.c).
- Instead, let's create a pointer p, a `dllink *` (a pointer to a dllink), make it point at one or other dllink, change the link thru the pointer *p, giving us allinone4.c:

```
dllink last = list->back;   // find the end of the list
dllink *p = last == NULL ? &(list->front) : &(last->next);
*p = new;                   // set EITHER list->front or last->next to new
```

- Here's our final refactored code:

```
dllist dl_build_list( int from, int to ) {
  dllist list = NEW(dllist);     // first, create an empty list
  assert( list != NULL );
  list->front = NULL;
  list->back = NULL;

  for( int i=from; i <= to; i++ ) {
    dllink new = NEW(dllink);   // create new element
    assert( new != NULL );
    new->value = i;
    new->next = NULL;
    printf("debug: make new node with value %d\n", new->value);

    // now link new onto the end of the list
    dllink last = list->back;   // find the end of the list
    dllink *p = last == NULL ? &(list->front) : &(last->next);
    *p = new;                   // set EITHER list->front or last->next to new
    new->prev = last;           // backlink prev(new) = old-last
    list->back = new;           // new is now the back of the list..
  }
  return list;
}
```

Other Functions

A `dl_free()` function to free a `dllist`:

```
void dl_free( dllist l ) {
        dllink next;
        for( dllink p = l->front; p != NULL; p = next) {
                next = p->next;
                free( p );
        }
        free( l );
}
```

Plus a `main()` function to test it all out, traversing the list forward through the list to print it out:

```
int main() {
        dllist l = dl_build_list( 10, 15 ); // create the list
        for( dllink p = l->front; p != NULL; p = p->next) { //
print the list out, traversing forward
```

```
                if( p != l->front ) putchar( ',' );
                printf( "%d", p->value );
        }
        putchar( '\n' );
        dl_free( l );
        return 0;
}
```

In an expanded version, we replace the list printout traversal with two traversals (traversing the list forward through the list from the front, and backward through the list from the back), flattening the elements we find during both traversals into a plain `int outdata[14]` array, and then printing that out:

```
int outdata[14];
int *op = outdata;
// traverse forward through the list, copying elements to outdata[]
for( dllink p = l->front; p != NULL; p = p->next) {
        printf("traverseF: %d\n", p->value);
        *op++ = p->value;
}
// traverse backward through the list, copying elements to
outdata[]
for( dllink p = l->back; p != NULL; p = p->prev) {
        printf("traverseB: %d\n", p->value);
        *op++ = p->value;
}
printf( "outdata[]: " );
for( int *p = outdata; p<op; p++ ) {
        if( p>outdata ) putchar( ',' );
        printf( "%d", *p );
}
putchar( '\n' );
```

When we compile and run, we see:

```
debug: make new node with value 10
debug: make new node with value 11
..
```

```
traverseF: 10
traverseF: 11
..
traverseF: 14
traverseF: 15
traverseB: 15
traverseB: 14
..
traverseB: 11
traverseB: 10
outdata[]: 10,11,12,13,14,15,15,14,13,12,11,10
```

Extending the program

Add forward and reverse traversal with callback functions

Here, we have added a traverser callback (pointer to function) type:

```
typedef void (*dl_trav_callback)( int, void * );
```

Then added forward and backward traversers:

```
void dl_traverse_backward( dllist l, dl_trav_callback cb, void
*state ) {
        for( dllink p = l->back; p != NULL; p = p->prev) {
                printf("traverseB: %d\n", p->value);
                (*cb)( p->value, state );
        }
}
void dl_traverse_forward( dllist l, dl_trav_callback cb, void
*state ) {
        for( dllink p = l->front; p != NULL; p = p->next) {
                printf("traverseF: %d\n", p->value);
                (*cb)( p->value, state );
        }
}
```

Finally, replace the explicit traversal code with calls to the above traverser functions (with the state data comprising a pointer to an `int *`. i.e. `&op` given `int *op`):

```
static void storeout( int value, void *extra ) {
        int **p = extra;
        *(*p)++ = value;
}
int main() {
        dllist l = dl_build_list( 10, 15 );
        int outdata[14];
        int *op = outdata;
        dl_traverse_forward(l, &storeout, &op );
        dl_traverse_backward(l, &storeout, &op );
        ...
}
```

We could turn our doubly linked list of integers into a doubly linked list of generic pointers

A Binary Search Tree Data Type

Let’s build a BST whose keys are arbitrary strings, which does not contain values, and focus only on adding keys

We should handle memory allocation for the keys - when allocating a new tree node, we duplicate the key string, using the library function `strdup()`

Let's start with our data type:

```
struct strbst;                  // forward definition
typedef struct strbst *strbst; // pure binary tree of (sorted) strings
struct strbst {
  char *key;                    // the key: a string
  strbst left, right;           // the left and right subtrees
};
```

As usual, we'll define functions for allocating memory:

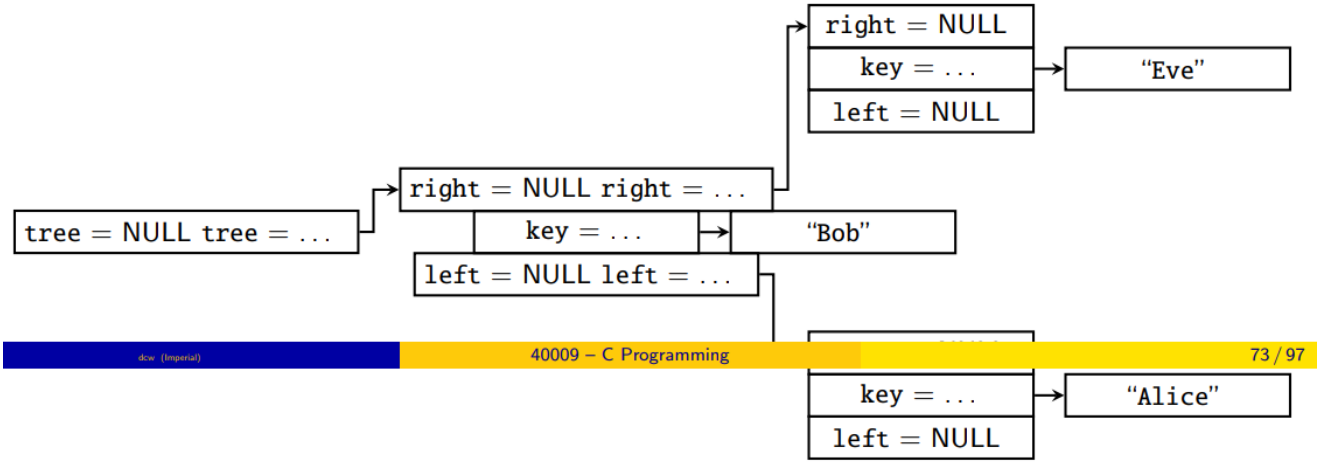
```
#define make_empty_strbst()  NULL

// strbst t = makenode( key );
//   Make a new bst node with <key>, duplicating the key.
//   Abort if any memory allocation fails.
static strbst makenode( char *key ) {
  strbst t = NEW( strbst );
  assert( t != NULL );
  key = strdup(key);        // key = strcpy( malloc(1+strlen(key), key )
  assert( key != NULL );
  t->key     = key;
  t->left    = NULL;
  t->right   = NULL;
  return t;
}
```

Adding a key to the Binary Tree: The simplest way to add a key to a BST is recursive (although iterative versions are possible, but tricky):

```
// t = add_strbst( t, key );
//   Add <key> to <t>, if it's not already present.
//   Abort if any memory allocation fails.
strbst add_strbst( strbst t, char *key ) {
  if( t == NULL ) {
    return makenode(key);
  }
  int cmp = strcmp( key, t->key );
  if( cmp == 0 ) {
    // already present
    return t;
  }
  if( cmp < 0 ) {
    t->left = add_strbst( t->left, key );
  } else {
    t->right = add_strbst( t->right, key );
  }
  return t;
}
```

Let's see the process of adding several keys to our BST diagrammatically:

```
strbst tree = make_empty_strbst();


strbst tree = make_empty_strbst();
tree = add_strbst(tree, "Bob");

strbst tree = make_empty_strbst();       tree = add_strbst(tree, "Alice");
tree = add_strbst(tree, "Bob");

strbst tree = make_empty_strbst();       tree = add_strbst(tree, "Alice");
tree = add_strbst(tree, "Bob");          tree = add_strbst(tree, "Eve");
```



Finding whether a key is present in a BST:

This function has a natural iterative structure, because at each stage we choose either the left or right subtree to continue searching in:

```
// bool present = in_strbst( t, key );
//   Return true iff <key> is in <t>.  Else return false.
bool in_strbst( strbst t, char *key ) {
  while( t != NULL ) {
    int cmp = strcmp( key, t->key );
    if( cmp == 0 ) return true;
    t = ( cmp<0 ) ? t->left : t->right;
  }
  return false;
}
```

In-Order Traversal of our BST:

This is naturally recursive:

```
// foreach_strbst( t, &kf, state );
//   Perform an in-order traversal of the tree), invoking <kf>
//   (the per-key callback function) for each key, passing it the
//   key and the <state>
void foreach_strbst( strbst t, strbst_key_f kf, void *state ) {
  if( t == NULL ) return;
  if( t->left != NULL ) foreach_strbst( t->left, kf, state );
  (*kf)( t->key, state );
  if( t->right != NULL ) foreach_strbst( t->right, kf, state );
}
```


Printing our BST out

Let's print all the keys out, comma-separated and surrounded by { ...}. Start by writing:

```
// print_strbst( b, out );
//   Print the strbst <b> to <out>, as an ordered key sequence,
//   separated by commas, and wrapped in '{'..'}'.
void print_strbst( strbst b, FILE *out ) {
  fprintf( out, "{ " );
  (void) rec_print( b, out, 0 );
  fprintf( out, " }" );
}
```

Then we need a helper function called `rec_print()` to do the rest of the work:

```
// int ip = rec_print( t, out, itemsprinted );
//   Print the strbst <t> to <out>, as an ordered key sequence,
//   separated by commas.
//   <itemsprinted> is the number of items already printed
//   Returns the number of items printed after printing this strbst.
static int rec_print( strbst t, FILE *out, int ip ) {
  if( t->left != NULL ) ip = rec_print( t->left, out, ip );
  if( ip>0 ) fputc( ',', out );
  fprintf( out, "%s", t->key );
  ip++;
  if( t->right != NULL ) ip = rec_print( t->right, out, ip );
  return ip;
}
```


Freeing our BST

Finally we recurse the tree and free everything (including the keys) – making sure to destroy parent nodes after their children:

```
// free_strbst( t );
// Free the given strbst <t>.
void free_strbst( strbst t ) {
        if( t->left != NULL ) free_strbst( t->left );
        if( t->right != NULL ) free_strbst( t->right );
        free( t->key ); // was strdup()ed, remember
        free( t );
}
```

Testing our BST

Our main() function in teststrbst.c constructs our BST in the obvious way, performing a series of membership tests at each stage:

```
strbst b = make_empty_strbst( );
member_tests( b, 0x0 );

b = add_strbst( b, "hello" );
member_tests( b, 0x1 );
...
```

The hex value is a small ad-hoc implementation of a (very small) set, determining whether or not the BST should contain each of a small number of known keys. See member_tests() for more details.
There are many more things we could with with our BST, but this gives you the basic flavour.

Extending the BST - storing `(key, value)` pairs

One obvious thing we could do is to add values to our BST – as we said earlier, BSTs can store (key,value) pairs, and thus be an implementation of mappings

The values could be some specific type or could be generic (void pointers)

The main change that would be necessary to do this is that (just like some earlier generic ASTs) we would need two structure types (and pointers to them) to represent the type – the BST that we have already seen would become a private internal binary tree, and the public BST type would become a structure that contains a binary tree (perfect use for an incomplete type) and several pointers to functions, where each function would implement an operation on the generic value – printing one out, freeing one etc

You will find such a generic BST implementation in example `adts/bst_string_k_generic_v`, check it out

Simulating objects

The combination of structs, pointers to structs, and function pointers within those structs, can also be used to emulate an OOP interface/class in C

Each function pointer in the struct needs to take a state parameter, typically a `void *`

This should be filled in with a pointer to the struct itself – this emulates the OOP concept of `self`

The struct of function pointers corresponds to the methods supported by the interface/class being emulated

A stream reading interface from the Vorbisfile library

```
typedef struct {
  size_t (*read_func)  (void *ptr, size_t size, size_t nmemb, void *datasource);
  int    (*seek_func)  (void *datasource, ogg_int64_t offset, int whence);
  int    (*close_func) (void *datasource);
  long   (*tell_func)  (void *datasource);
} ov_callbacks;
```