A Container Holds Other Data Structures

What's a container?
- a data structure
- containing other data structures
- with specific access capabilities.

Examples include
- linked lists,
- heaps (the type used for MP9), and
- dynamically-sized arrays.

Example: a Singly-Linked List Containing “Things”

For simplicity,
- consider a singly-linked list
- containing “things.”

How do the list elements hold “things?”

Adding Containers to Structures Implies More Code

When we studied lists,
- we added a next field
- to a data structure
- to create a linked list.

And said: just
- add more fields
- to create multiple lists.

But to do so, we must write code for each list separately.
List Elements Can Point to the “Things”

To avoid writing list code repeatedly,
◦ we can create a list element structure
◦ with a data pointer (void*),
◦ then build a list with list nodes.

Other Lists Can Point to the Same Set of “Things”

For another list, use more list elements!

Extra Level of Indirection: Simpler Code, But Slower

Using an extra level of indirection
◦ (a pointer to a structure with a pointer)
◦ implies that we write the list code once
◦ (all lists look the same to the code!)
◦ but use requires more memory accesses.*

Simpler Code Can Be Achieved in Another Way

There is another option:
◦ to write the list code once,
◦ the next field must point to a list element structure,
◦ but that structure may be part of a larger structure.

*Also note that some container properties may be affected. Removal from a doubly-linked list built in this way is not fast if one has only a “thing” pointer, for example.
Including List Element is Faster, But Allows Just One List

In other words,
◦ create a list element structure
◦ consisting of a next field (only), and
◦ include a list element structure
◦ as first element of the “thing” structure.
For list code, a “thing” is just a list node.
List nodes can be turned into “things” (the address is the same)!
But can only have one list for any “thing.”

Example: Cyclic, Doubly-Linked List with a Sentinel

Let’s build an example container:
◦ cyclic, doubly-linked list with a sentinel
◦ using list element inclusion
(rather than extra pointers).
In particular,
◦ a double_list_t includes pointers
to next and prev double_list_t’s, and
◦ first field of “thing” is a double_list_t.

A List Node is Simple: Just Two Pointers

typedef struct double_list_t double_list_t;
struct double_list_t {
    double_list_t* prev;
    double_list_t* next;
};

How do we initialize a list?

List Variables Can be Statically or Dynamically Initialized

List variables can be initialized statically:
// this variable is the sentinel
static double_list_t my_list = {
    &my_list, &my_list};

But dynamically-allocated lists must be initialized at runtime.
Initialize Dynamically by Pointing Sentinel to Itself

Let’s write a routine for initialization:

```c
void dl_init (double_list_t* head)
{
    head->prev = head->next = head;
}
```

List Insertion Requires the Head and the New Element

To insert an element into a list, we need

- a pointer to the list head and
- a pointer to the new element.

For example, we might write:

```c
void dl_insert (double_list_t* head,
                double_list_t* elt);
```

Structures for Insertion Include List and Element

```c
void dl_insert (double_list_t* head,
                double_list_t* elt);
```

So we have...

First Two Insertion Steps Change New Element

```c
elt->next = head->next; // step 1
elt->prev = head;      // step 2
```
Second Two Insertion Steps Change List

head->next->prev = elt;  // step 3
head->next = elt;        // step 4

Complete Code for List Insertion

Together, we have:

```c
void dl_insert (double_list_t* head,
                double_list_t* elt)
{
    elt->next = head->next;
    elt->prev = head;
    head->next->prev = elt;
    head->next = elt;
}
```

The “Thing” Must Have a double_list_t Field First

But what IS the “thing”?

Any data structure, as follows:

```
struct thing_t {
    double_list_t dl;
    // the rest of the structure
};
```

double_list_t must be first in structure.

To Insert, Use the Address of the double_list_t Field

Once we create a list and a thing:

```c
double_list_t my_list = {
    &my_list,  // my_list
    &my_list   // my_list
};
struct thing_t my_thing;
```

We can insert the thing into the list as follows:

```c
dl_insert (&my_list, &my_thing.dl);
```

Same address as &my_thing.
Pitfall: Not Putting the `double_list_t` First

What happens if the `double_list_t` is not first in “thing?”

```c
struct thing_t {
    // some stuff
    double_list_t dl;
    // the rest of the structure
};
```

`&my_thing->dl` points here.

```
thing
prev
more
thing
```

`double_list_t` MUST be First in Thing!

What happens if the `double_list_t` is not first in “thing?”

```c
struct thing_t {
    // some stuff
    double_list_t dl;
    // the rest of the structure
};
```

`&my_thing->dl` points here.

```
thing
prev
more
thing
```

Cannot convert `double_list_t*` to `struct thing_t*` (need to subtract unknown amount from the pointer).

List Removal Requires Only the Element to Remove

To remove an element from a list, we only need a pointer to the element to be removed.

For example, we might write:

```c
void dl_remove (double_list_t* elt);
```

Structures for Removal Include Only the Element

```c
void dl_remove (double_list_t* elt);
```

So we have...
Two Steps for Removal from List

```
elt->prev->next = elt->next;  //step 1
elt->next->prev = elt->prev;  //step 2
```

Can Also Provide Other Access Methods

We can also provide other access methods.

For example, a routine to find the first element in a list.

```
// Returns a pointer to the first element in a list.
// Returns a pointer to the first element in a list, or NULL for
// an empty list.
void* dl_first (double_list_t* head);
```

One Line Needed to Find the First Element in a List

Such a routine is not hard to write:

```
// Returns a pointer to the first element in a list, or NULL for
// an empty list.
void* dl_first (double_list_t* head)
{
    return (head == head->next ?
            NULL : head->next);
}
```

Can We Provide a Way to Iterate Over “Things”? 

What about iterating over the list elements?

Let’s say that we want
* to execute some code
* for every “thing” in a list.

Again, we want to
* avoid exposing the details of our list implementation, and
* avoid requiring many copies of iteration control code.
Can we write a function to perform iteration?

Yes, by placing the code to execute for each “thing” in a callback!

What signature should we use for the callback?

First, we need:
- a pointer to the “thing.”
- Let’s use a `void*`.

To allow specialization, we also need:
- a pointer to other arguments,
- also a `void*`.

What should the callback return?

If all we want to do:
- is iterate over all things,
- return type `void` is fine.

What else might we want to do?

1. Find a specific “thing”
   (and possibly remove it from the list).
2. Remove a set of “things” from the list
   (remove and keep going).
3. Free a set of “things” from the list
   (remove, free, and keep going).
Here we have design choices.

Our choice:
- one **can safely change the list** in the callback function,
- **EXCEPT** that one **cannot remove the “thing”** passed to the callback from the list.

(The alternative choices can be confusing to use.)

---

**Design to Allow Changes to List During Callback**

---

**Callback Returns a Number Saying What to Do**

So, again, what should the callback return?

An enumerated constant!

Let’s call it `dl_execute_response_t`.

---

**Meanings of Possible Callback Return Values**

```c
typedef enum {
    DL_CONTINUE,             /* Keep going. */
    DL_STOP_AND_RETURN,      /* Return this “thing.” */
    DL_REMOVE_AND_CONTINUE,  /* Remove “thing” and continue with next one. */
    DL_REMOVE_AND_STOP,      /* Remove “thing” and return it. */
    DL_FREE_AND_CONTINUE     /* Free “thing” and continue with next one. */
} dl_execute_response_t;
```

---

**Meanings of Possible Callback Return Values**

```c
typedef enum {
    DL_CONTINUE,             /* Keep going. */
    DL_STOP_AND_RETURN,      /* Return this “thing.” */
    DL_REMOVE_AND_CONTINUE,  /* Remove “thing” and continue with next one. */
    DL_REMOVE_AND_STOP,      /* Remove “thing” and return it. */
    DL_FREE_AND_CONTINUE     /* Free “thing” and continue with next one. */
} dl_execute_response_t;
```
Iteration Callback Takes Two `void*`, Returns a Number

And here's the definition for the callback function:

```c
typedef dl_execute_response_t
  (*dl_execute_func_t)
  (void* dl, void* arg);
```

Iteration Requires Three Arguments

Now we can also write the function signature for iteration:

```c
void* dl_execute_on_all
  (double_list_t* head,
   dl_execute_func_t func,
   void* arg);
```

Iteration Returns a Pointer to a “Thing”

Now we can also write the function signature for iteration:

```c
void* dl_execute_on_all
  (double_list_t* head,
   dl_execute_func_t func,
   void* arg);
```

Local Variables for our Iteration Function

And here's the function:

```c
double_list_t* dl;
double_list_t* remove;
dl_execute_response_t result;
```
For Every “Thing,” Call the Callback and Take Action

Loop over all things.

for (dl = head->next; head != dl; dl = dl->next) {

result = (*func) (dl, arg);

switch (result) {

Call the callback function.

Take action based on callback’s response.

Cases to Return “Thing” After Possibly Removing It

switch (result) {

Remove the “thing”...

... and return it! (no break)

This case returns “thing” without removing.

Cases to Remove “Thing” and Continue (Maybe Freeing)

case DL_REMOVE_AND_CONTINUE:
case DL_FREE_AND_CONTINUE:

Copy “thing” to remove.

Remove “thing” from list.

Loop update reads this element’s next.

If requested, free “thing” (other data can be freed by callback).
If loop ends, return NULL.

Case is Based on Loop

```c
default:
    break;

}  // end of switch

// end of loop over "things"
return NULL;

If loop ends, return NULL.
```

That's all!

The code is part of Lab 14.