comp10002
Foundations of Algorithms

Semester Two, 2025

Structures and Dynamic Memory

Lecture slides prepared by Alistair Moffat

Overview

Structures (Chapter 8)

Dynamic memory (Chapter 10)

Lists, stacks, and queues

Trees

Dictionaries

Polymorphism

Chapter 8 – Concepts

In an array, homogeneous-typed data is aggregated, with individual elements identified by ordinal position and accessed using the pointer dereference operator `*` and `[]`.

In a `struct`, heterogeneous-typed data is aggregated, with individual elements identified by component name, and accessed via the "`.`" selection operator.

The operator `->` allows direct access to the components of a structure identified by a pointer.

Arrays and structures can be composed to make hierarchical data representations.

Structures – Example

```
typedef struct {
    int dd, mm, yyyy;
} date_t;

typedef struct {
    int mm, ss;
} time_t;

typedef char name_t[NAMLEN+1];

typedef struct {
    int    lec_num;        // eg, 07
    char   lec_seq;        // eg, 'a'
    name_t presenter;      // eg, "Alistair Moffat"
    date_t created;        // eg, 24 08 2021
    date_t released;       // eg,  8 09 2021
    time_t duration;       // eg, 26 58
} video_t;

video_t one_video;
video_t allvideos[MAXVID];
int     nvideos;          // buddy variable for allvideos[]
```

Structures – Example

The structure `one_video` refers to a single video recording.

And `one_video.created` is the date the video was created, and is itself a structure of type `date_t`; and `one_video.created.yyyy` is the year it was recorded.

Whole structures can be *assigned*: `allvideos[nvideos++] = one_video` adds that package of information to the (extended by one) collection `allvideos`.

Structures – Example

If an array of structures is passed to a function, it is passed (like all arrays) as a pointer:

```
int
total_minutes(video_t V[], int nv) {
    int mm=0, ss=0;
    int i;
    for (i=0; i<nv; i++) {
        mm += V[i].duration.mm;
        ss += V[i].duration.ss;
    }
    return mm + (int)(0.5+1.0*ss/SEC_PER_MIN);
}
```

But when a single structure is passed, it is *copied* (like all variables). So it is usual to pass structure *pointers* to functions, to avoid copying dozens or hundreds of bytes.

Structures – Example

Passing pointers allows changes to structure *components*:

```
void
set_release_date(video_t *v, int dd, int mm, int yyyy) {
    v->released.dd = dd;
    v->released.mm = mm;
    v->released.yyyy = yyyy;
    return;
}
```

Compare that with:

```
void
set_presenter_silly(video_t v, char *name) {
    /* pointless code, because local changes will be
       discarded upon return, even though v.presenter
       is an array */
    strncpy(v.presenter, name, NAMLEN);
    return;
}
```

Structures – Example

Structures can be created and *returned*, and can also be initialized on declaration:

```
time_t
total_time(video_t V[], int nv) {
    time_t total = {0,0};
    int i;
    for (i=0; i<nv; i++) {
        total.mm += V[i].duration.mm;
        total.ss += V[i].duration.ss;
    }
    while (total.ss>=SEC_PER_MIN) {
        total.mm += 1;
        total.ss -= SEC_PER_MIN;
    }
    return total;
}
```

Note that it makes no sense to try and return a *pointer* to a local structure, because the underlying variable is short-lived.

Structures – Comparison functions

To compare two structures a comparison function should always be written:

```
int
cmp_video_label(video_t *v1, video_t *v2) {
    if (v1->lec_num < v2->lec_num) return -1;
    if (v1->lec_num > v2->lec_num) return +1;
    if (v1->lec_seq < v2->lec_seq) return -1;
    if (v1->lec_seq > v2->lec_seq) return +1;
    return 0;
}
```

The return value is –ve if the first argument is being regarded as "smaller", 0 if they can appear in any order, and +ve if the second argument is the one that is "smaller".

Structures – Comparison functions

For *reverse* date ordering (most recent first):

```
int
cmp_reverse_date(date_t *d1, date_t *d2) {
    if (d1->yyyy > d2->yyyy) return -1;
    if (d1->yyyy < d2->yyyy) return +1;
    if (d1->mm   > d2->mm  ) return -1;
    if (d1->mm   < d2->mm  ) return +1;
    if (d1->dd   > d2->dd  ) return -1;
    if (d1->dd   < d2->dd  ) return +1;
    return 0;
}
```

Such functions will become important in Chapter 10.

Structure and array operations

There are fundamental differences between arrays and structures. An array is pointer, a structure is an object.

	Array	*Struct*
Assigned (`=`)	No	Yes
Compared (`==`)	Yes (as pointer)	No
Argument to function	Yes (as pointer)	Yes (copied)
Returned from function	No (but see Ch 10)	Yes (copied)
Take address of (`&`)	No (is already)	Yes
Use as pointer (`*`, `[]`)	Yes	No

An array of structures behaves as an array. A structure that has an array as an element still behaves as a structure.

Chapter 8 – Program examples

Structures can be nested, and used in struct/array/struct/array hierarchies.

- `struct.c`
- `nested.c`

Chapter 8 – Exercises

People have titles, a given name, a middle name, and a family name, all of up to 50 characters each. People also have dates of birth (dd/mm/yyyy), dates of marriage and divorce (as many as 10 of each), and dates of death (with a flag to indicate whether or not they are dead yet). Each date of marriage is accompanied by the name of a person. Assuming that people work for less than 100 years each, people also have, for each year they worked, a year (yyyy), a net income and a tax liability (both rounded to whole dollars), and a date when that tax liability was paid.

Countries are collections of people. Australia is expected to contain as many as 30,000,000 people; New Zealand as many as 6,000,000 people.

Chapter 8 – Exercises

Exercise 1

Give declarations that reflect the data scenario that is described.

Exercise 2

Write a function that calculates, for a specified country indicated by a pointer argument (argument 1) with a number of persons in it (argument 2), the average age of death. Do not include people that are not yet dead.

Chapter 8 – Exercises

Exercise 3

Write a function that calculates, for the country indicated by a pointer argument (argument 1) with a number of persons in it (argument 2) the total taxation revenue in a specified year (argument 3).

Now that you see the processing mode implied by this exercise, do you want to go back now and revise your answer to Exercise 1? If you did, would you need to alter your function for Exercise 2 at all?

Chapter 8 – Summary

Key messages:

- Structures provide data abstraction in the same way that functions provide execution abstraction.
- When composed with arrays, structures allow complex data hierarchies to be represented
- Correct use of structures adds flexibility to programs – additional data elements can be added should the need arise without altering functions that don't use the extra data.

Chapter 10 – Concepts

New tracts of memory, sized according to run-time values returned by `sizeof()`, can be requested via the `malloc()`.

The allocated memory is manipulated via a returned pointer.

When no longer required, the memory is handed back via `free()`, and becomes available to future `malloc()` requests.

Once allocated, memory segments that have already been allocated can be resized using `realloc()`.

Chapter 10 – Concepts

Recursive `struct` types include pointers of their own type.

With one dimensional recursive structures, lists, queues, and stacks can be created.

With two dimensional recursive structures, trees and tries can be constructed.

Higher dimensional data structures such as undirected and directed graphs can also be made.

Memory allocation – Standard recipe

To allocate an array for `n` items each of type `type_t`:

```
int n;
type_t *tptr;

/* determine how big the array needs to be today */
n = ... ;

/* and ask for the right amount of space */
tptr = (type_t*)malloc(n*sizeof(*tptr));
assert(tptr);

/* then use tptr[0..n-1] as an array in the usual manner */

/* and give it back when finished */
free(tptr);
tptr = NULL;
```

Memory allocation

Some warnings:

- Always use `sizeof()`, don't hard-code sizes
- Always test (or `assert()`) the pointer that is returned
- Remember that garbage collection is your responsibility
- Match every `malloc()` with a corresponding `free()` (if not, will give rise to a *memory leak*)
- Always set the pointer to `NULL` after a `free()`, to prevent improper re-access
- Use `realloc()` to grow multiplicatively, not additively.

Chapter 10 – Program examples (Part I)

- `sizeof.c`
- `malloc.c`
- `realloc.c`

Chapter 10 – Exercises

Exercise 4

Write a function `char *string_dupe(char *s)` that creates a copy of the string `s` and returns a pointer to it.

Exercise 5

Write a function `char **string_set_dupe(char **S)` that creates a copy of the set of string pointers `S`, assumed to have the structure of the set of strings in `argv` (including a sentinel pointer of `NULL`), and returns a pointer to the copy.

Chapter 10 – Exercises

Exercise 6

Write a function `void string_set_free(char **S)` that returns all of the memory associated with the duplicated string set `S`.

Exercise 7

Test all three of your functions by writing scaffolding that duplicates the argument `argv`, then prints the duplicate out, then frees the space.

(What happens if you call `string_set_free(argv)`? Why?)

Linear linked structures

Simple idea: define a `struct` type that includes a pointer to itself (rather than a pointer to, say, a string):

```
typedef struct node node_t;

struct node {
    data_t data;
    node_t *next;
};
```

Note the need for the forward declaration – the type `node_t` is *declared* before it is *defined*, the same as is done via function prototypes.

Linear linked structures

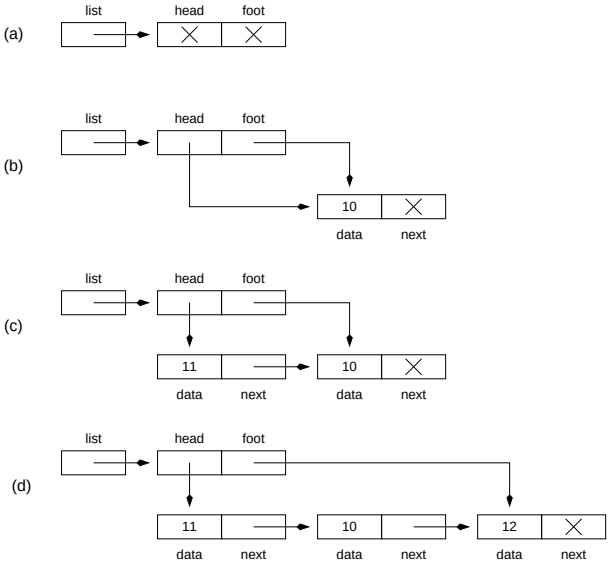
So, now what? Build linked lists is what. Define another type to hold the first pointer in the chain, together with a pointer to the end of the chain:

```
typedef struct {
    node_t *head;
    node_t *foot;
} list_t;
```

In each list a sequence of elements of type `data_t` are threaded together using a chain of pointers, with a "handle" of type `list_t`.

Last node in the chain has a `NULL` pointer.

Linear linked structures



Linear linked structures

Given a list handle, easy to sequentially process each element in the order they appear in the list:

```
void
process_every_data_item(list_t *list) {
    assert(list);
    p = list->head;
    while (p) {
        process(p->data);
        p = p->next;
    }
}
```

Linear linked structures

Also easy to push a new element to the front of the list.

```
list_t
*insert_at_head(list_t *list, data_t value) {
    node_t *new;
    new = (node_t*)malloc(sizeof(*new));
    assert(list && new);
    new->data = value;
    new->next = list->head;
    list->head = new;
    if (list->foot==NULL) {
        /* this is the first insertion into the list */
        list->foot = new;
    }
    return list;
}
```

It is standard for such functions to return a pointer to the manipulated object, even if it didn't get moved.

Linear linked structures

And if we want to pop the front element off the list and discard it:

```
list_t
*get_tail(list_t *list) {
    node_t *oldhead;
    assert(list!=NULL && list->head!=NULL);
    oldhead = list->head;
    list->head = list->head->next;
    if (list->head==NULL) {
        /* the only list node just got deleted */
        list->foot = NULL;
    }
    free(oldhead);
    return list;
}
```

Stacks and queues

The other option is to add new items at the tail of the list.

The `foot` pointer can be used to implement an $O(1)$-time `insert_at_foot()` operation. (Try it!)

But it is *not* possible to quickly delete the last item in the list without adding further pointers.

- `listops.c`
- `listeg.c`

Stacks and queues

If insertions are at the foot of the list and removals from the head, the structure is a queue, or first-in first-out (FIFO) structure.

If insertions and removals are both at the head of the list, the structure is a stack, or last-in first-out (LIFO) structure.

Stacks and queues are fundamental data structures that are used in a wide range of algorithms.

Chapter 10 – Exercises

Exercise 8

Stacks and queues can also be implemented using an array of type `data_t`, and associated state variables, bundled together into a struct. Give functions for `make_empty_stack()` and `push()` and `pop()` in this representation.

Exercise 9

Suppose that insertions and extractions are required at both head and foot. Add a second pointer to each node (to make a doubly-linked list) and then rework the previous functions.

Binary search trees

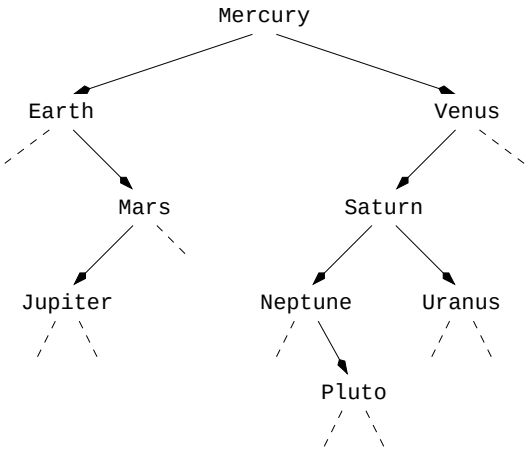
Next step – nodes with two pointers that act independently:

```
typedef struct node node_t;

struct node {
    void *data;
    node_t *left;
    node_t *rght;
};
```

Note also (as a further change) that `data` is now being stored via an anonymous pointer. This `treeops` library is going to be polymorphic.

Binary search trees – Example



Binary search trees – Analysis

If input data is a random permutation, average depth of a leaf will be $O(\log n)$.

So *average* search cost will be $O(\log n)$ key comparisons, whether successful or unsuccessful.

But in worst case, tree is a stick, and search takes $O(n)$ key comparisons. (What sequence?) Not very palatable.

Can we *randomize*? Yes, but only if all of the items to be inserted are available in advance.

Dictionary abstract data type

If we have an algorithm that requires the operations insert() and search(), now have several possible structures:

	Insert	Search
Unsorted array	$O(1)$ wc	$O(n)$ wc
Sorted array*	$O(n)$ wc	$O(\log n)$ wc
Linked list	$O(1)$ wc	$O(n)$ wc
BST	$O(n)$ wc, $O(\log n)$ avg	$O(n)$ wc, $O(\log n)$ avg

Analysis is based on key comparisons, which might or might not be $O(1)$ time each, and averaging assumes random insert/search keys.

* Note the error in this row in Table 12.2 in the textbook. Ooops.!

Binary search trees – Implementation

To allow a polymorphic (type-free) implementation, the comparison function must be captured at the time the tree is created, rather than being passed in to every tree manipulation function.

```
typedef struct {
    node_t *root;
    int (*cmp)(void*,void*);
} tree_t;
```

Binary search trees – Implementation


```
tree_t
*make_empty_tree(int func(void*,void*)) {
    tree_t *tree;

    tree = (tree_t *)malloc(sizeof(*tree));
    assert(tree!=NULL);

    /* initialize tree to empty */
    tree->root = NULL;

    /* save the supplied function pointer */
    tree->cmp = func;

    return tree;
}
```

Abstraction at its best: with these declarations, a program can maintain multiple trees with multiple types stored, and in any desired item ordering.

Chapter 10 – Program examples (Part II)

- `funcpoint.c`
- `funcarg.c`
- `callqsort.c`
- `treeops.h`
- `treeops.c`
- `treeeg.c`

Looking forward

Is it possible to have a tree-based data structure in which the worst-case cost is $O(\log n)$ key comparisons for search and insert (and delete)?

Yes, but you will need to come back and take another subject to find out how.

Ok, well, is it possible to have a dictionary data structure that takes $O(1)$ average-case key comparisons per operation?

Yes, but you will need to come back in a couple of weeks.

Dynamic data structures – Summary

Dynamic memory allows run-time construction of linked data structures.

Lists and trees are very powerful algorithmic techniques.