COMP10002

Semester One, 2017

Structures and Dynamic Memory

Overview

Structures (Chapter 8)

Dynamic memory (Chapter 10)

Lists, stacks, and queues

Trees

Dictionaries

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 char acc_num_t[ACCLEN+1];

typedef struct {
    accnum_t accnum;
    date_t   first_deposited;
    date_t   last_interest_paid;
    double   current_amount;
    double   current_interest_rate;
    int      interest_interval;
    accnum_t interest_paid_to;
    date_t   next_interest_due;
    double   interest_accrued;
} termdeposit_t;

termdeposit_t new_deposit, alldeposits[MAXTD];
int num_termdeposits;
```

Structures – Example

Then `new_deposit` is a package of information about one term deposit.

And `new_deposit.current_amount` is the current balance of that one deposit.

And `new_deposit.next_interest_due` is a structure of type `date_t`; `new_deposit.next_interest_due.yyyy` is the year component of it.

And `alldeposits[num_termdeposits++] = one_deposit` adds that information to the (increased in size by one) collection of such information.

Structures – Example

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

```
double
total_liabilities(termdeposit_t all[], int num) {
    double sum=0.0;
    int i;
    for (i=0; i<num; i++) {
        sum += all[i].current_amount;
        sum += all[i].interest_accrued;
    }
    return sum;
}
```

But when a structure is passed, it is *copied*. So it is usual to pass structure *pointers* to functions.

Structures – Example

```
void
td_daily_update(termdeposit_t *d, date_t *today) {
    d->interest_accrued =
        d->current_amount *
        d->current_interest_rate *
        calculate_year_frac(today, &(d->last_interest_paid));
    if (cmp_dates(&(d->next_interest_due),
            today) != 0) {
        /* no action required */
        return;
    }
    pay_to_account(d->interest_paid_to,
        d->interest_accrued);
    d->interest_accrued = 0.0;
    d->last_renewed = *today;
    add_to_date(&(d->next_interest_due),
                d->interest_interval);
    return;
}
```

Structure and array operations

There are some 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	Yes (as pointer)	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 behaves as a structure.

Chapter 8 – Program examples

- `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, can be requested via the function `malloc()`.

The allocated memory is manipulated via a pointer variable.

Amounts that have already been allocated can be resized using function `realloc()`.

When no longer required, the memory can be handed back via the function `free()`.

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`:

```
type_t *tptr;
/* figure how big the array needs to be */
n = ... ;
/* and ask for the right amount of space */
tptr = (type_t*)malloc(n*sizeof(*tptr));
if (!tptr) {
    printf("Error: no memory available\n");
    exit(EXIT_FAILURE);
}
/* or */
assert(tptr);
/* then */
do stuff with *tptr and/or tptr[0..n-1]
/* and finally */
free(tptr);
tptr = NULL;
```

Memory allocation

Some warnings:

- Always use `sizeof()`, don't hard-code sizes
- Always test 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*)
- Then always set the access pointer to `NULL`, 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.

Sequences of elements of type `data_t` are threaded together in a chain of pointers.

Last item in chain has a `NULL` pointer.

If `p` is a pointer to such a chain, can sequentially process each element in the chain.

And if `p` is a pointer to such a chain, can push a new element into the front of the chain.

Linear linked structures


```
void
process_each(node_t *p) {
    while (p) {
        process(p->data);
        p = p->next;
    }
}

node_t
*push(data_t stuff, node_t *p) {
    node_t *new;
    new = (node_t*)malloc(sizeof(*new));
    assert(new);
    new->data = stuff;
    new->next = p;
    return new;
}
```

Linear linked structures

Similarly, if `p` is a pointer to a non-empty chain, can `pop` the front element off the chain and discard it:

```
node_t
*pop(node_t *p) {
    node_t *old;
    assert(p);
    old = p;
    p = p->next;
    free(old);
    return p;
}
```

In this form, it *must* be used as `list = pop(list)` and not `newlist = pop(oldlist)`. (Why?)

Linear linked structures

Or, if the data from the front of the list is to be returned and the list shortened (take a deep breath):

```
data_t
pop(node_t **p) {
    node_t *old;
    data_t stuff;
    assert(p && *p);
    old = *p;
    stuff = (*p)->data;
    (*p) = (*p)->next;
    free(old);
    return stuff;
}
```

which is used via `front_data = pop(&list)`, with `list` holding a different value after the call than before.

Stacks and queues

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

Could traverse list to reach the insertion point. But better to maintain another level of abstraction that keeps pointers to the first and last item in the list; this is the purpose of the `list_t` type.

- `listops.c`

Stacks and queues

If insert at tail and extract from head, have a queue, or a first-in first-out (FIFO) structure.

If insert and head and also extract from head, have 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. They allow data to be processed systematically in orders other than it was received in.

Chapter 10 – Exercises

Exercise 8

Stacks and queues can also be implemented using an array of type `data_t`, and static variables. 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. How can `delete_foot()` be implemented efficiently? (Hint, can a second pointer be added to each node?)

Binary search trees

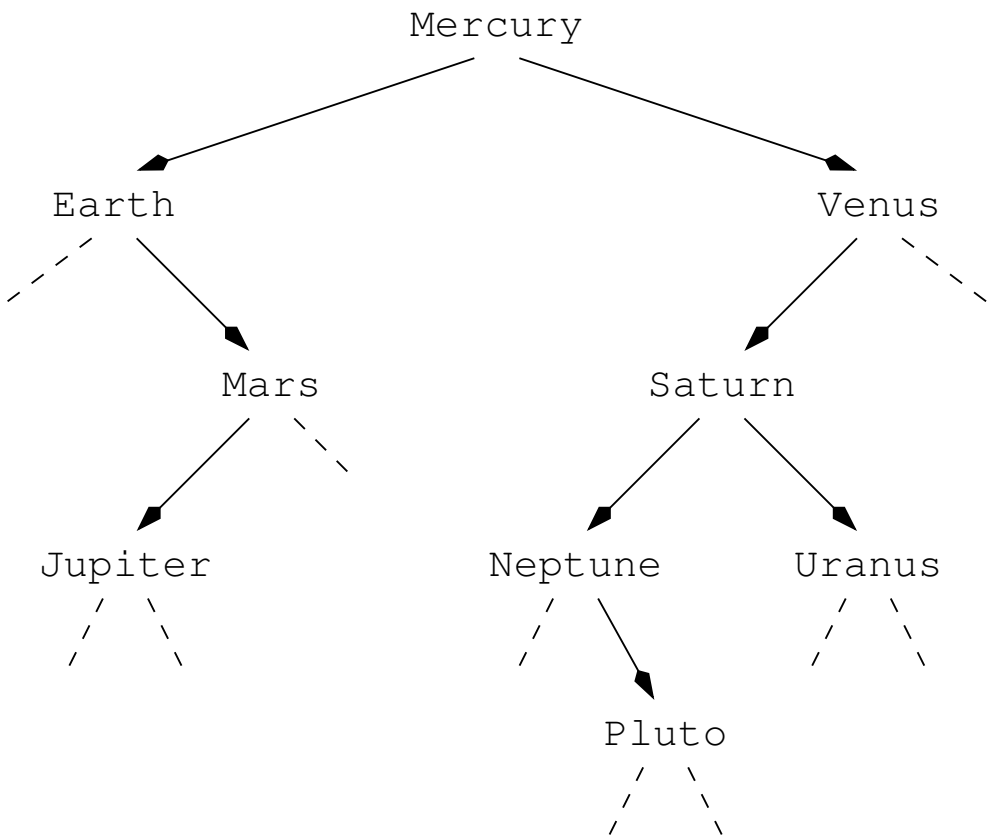
Next step – nodes with two pointers:

```
typedef struct node node_t;

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

Note also (as an independent change) that `data` is stored via an anonymous pointer. This library is now *polymorphic*.

Binary search trees – Example



Binary search trees

To allow type-free functions, the comparison function must also be polymorphic, and is 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


```
tree_t
*make_empty_tree(int func(void*,void*)) {
    tree_t *tree;
    tree = malloc(sizeof(*tree));
    assert(tree!=NULL);
    /* initialize tree to empty */
    tree->root = NULL;
    /* and save the supplied function pointer */
    tree->cmp = func;
    return tree;
}
```

Abstraction at its best: with these declarations, can have one tree of strings in ascending order, another of some data type in descending order.

Chapter 10 – Program examples (Part II)

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

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	$O(n)$ wc
BST	$O(\log n)$ ac	$O(\log n)$ ac

Note that all counts are of key comparisons, which might or might not be $O(1)$ time each.

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.