CS2850 Operating System Lab

Week 7: Linked Lists

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

Structures

Linked lists

Implementing linked lists in C

Doubly-linked and circular lists

Structures

Structures help *organize heterogeneous* data.

A structure is a *collection* of variables of *possibly different types*.
The variables are called structure members.

A structure has a single *type-name*, called the structure tag.

Defining a `struct`

Structure are defined *outside* `main` by specifying the type of each struct member, e.g.

```
struct structureTag{
   int member1Name;
   char *member2Name;
};
```

Objects of type `struct structureTag` can be declared *inside* `main` as usual, e.g.

```
int main() {
  int i;
  struct structureTag myStruct;
  ...
}
```

Accessing the `struct` members

There are *two equivalent ways* of accessing struct members, e.g.

```
myStruct.member1Name = 1;
*(myStruct.member2Name) = 'h';
```

or

```
(&myStruct)->member1Name = 1
*((&myStruct)->member2Name) = 'h'
```

Note: the *arrow notation* applies to the structure address.

Global operations

You can copy or assign structures *as a unit*, e.g.

```
struct structureTag myStruct2 = myStruct;
```

You can get the address of the entire `struct` with &, e.g.

```
struct structureTag *pointerToMyStruct= &myStruct;
```

You can compute the size of a structure with `sizeof`, e.g.

```
int sizeOfMyStruct = sizeof(struct structureTag);
```

Illegal operations

You can *not* compare two structures, e.g. you can *not* write

```
if(myStruct1 == myStruct2) {...}
```

You can *not* perform arithmetic operations between structures, e.g. you can *not* write

```
myStruct1 = myStruct1 + myStruct2
```

Why not?

Structures are composite objects and handling them *as a unit* is not always allowed.

Note: all this *does not apply* to single members, e.g. you can have

```
if(myStruct1.member2Name == (&myStruct2)->member2Name) {
    myStruct1.member1Name = myStruct2.member1Name + 1;
    ((&myStruct2)->member1)++;
}
```

Example (1)

```
#include <stdio.h>
#define MAX 100
struct word{
  int length;
  char s[MAX];
};
int main() {
  struct word helloWorld;
  char *s = "hello, world!";
  helloWorld.length = 0;
  while (*(s + helloWorld.length) != '\0') {
    *(helloWorld.s + helloWorld.length) = *(s + helloWorld.length);
    (&helloWorld)->length++;
  }
  *(helloWorld.s + helloWorld.length) = '\0';
  struct word myStruct = helloWorld;
  printf("myStruct.s=%s\n", myStruct.s);
  printf("myStruct.length=%d\n", myStruct.length);
  printf("sizeof(myStruct)=%lu\n", sizeof(myStruct));
}
```

Example (2)

The program above defines, initializes, copies and prints the content of a struct.

The output is

```
myStruct.s=hello, world!
myStruct.length=13
sizeof(myStruct)=104
```

Why do you need to initialize `myStruct.s` "*element by element*"?

Size of a struct

The size in bytes of structure may depend on the order you declare the struct members, e.g. in

```
#include <stdio.h>
#define MAX 4
struct word1{
  char c;
  int v[MAX];
  char c2;
};
struct word2{
  int v[MAX];
  char c;
  char c2;
};
int main() {
  struct word1 w1;
  struct word2 w2;
  printf("sizeof(w1)=%lu\n", sizeof(w1));
  printf("sizeof(w2)=%lu\n", sizeof(w2));
}
```

Linked lists of structures

Linked lists of structures are also called self-referential structures.

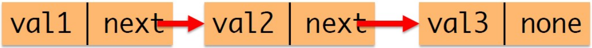
Each node in the list is a *pre-defined* object of type `struct` with *at least* two members:

1. the *value* of the node and
2. a *reference* to the next node in the list.

Implementation

The *reference to the next node* is a pointer to a `struct` object of the *same type*, e.g.

```
struct node {
    int val;
    struct node *next;
};
```



The last element in the list is a null pointer, `NULL`, so you can know when you have reached the end of the list.

Dynamical lists

Linked lists are useful if you *do not know* the number of nodes in advance.

The idea is to create new nodes when new data points *arrive*.

Creating a new node requires allocating a new memory slot of size `sizeof(struct node)`.

The total amount of memory needed is *unknown* at compile time, i.e. you need to allocate nodes dynamically (using `malloc`).

The first node

Dynamic linked lists are built iteratively by linking each new node to the *head of the list*.

A new node is declared and initialized as

```
struct node *pNode = malloc(sizeof(node));
pNode->val = someValue;
pNode->next = NULL;
```

The head

A pointer, `head`, is needed to keep track of *where is the list head*.

Declare it as

```
struct node *head = NULL;
```

and use it to store the address of the list head.

When the list consists of a single node you set

```
head = pNode
```

New nodes

As a new value, e.g. `newValue` $= 10$, arrives you need to

- allocate memory for a new node

```
struct node *cur = malloc(sizeof(struct node));
```

- store the new value in the node

```
cur->val = newValue;
```

- link the new node to the head and move the head to the new node

```
cur->next = head;
head = cur;
```

Creating the list

Initialize a list for storing a *variable number* of integers (and a 0 in the first node).

```
//headers
struct node{
        int val;
        struct node *next;
};
int main() {
  int n = getchar() - '0';
  struct node *head = NULL;
  struct node *cur = malloc(sizeof(struct node));
  cur->val = 0;
  cur->next = head;
  head = cur;
}
```

Iterations

Iteratively create `n` new nodes and store the first `n` integers into them.

```
for (int i=0; i < n; i++) {
  struct node *cur = malloc(sizeof(struct node));
  cur->val = (i+1);
  cur->next = head;
  head = cur;
}
```

Printing the list

Print information about the nodes (from the last node, i.e. the head, to the first).

```
struct node *cur = head;
for (int i=0; i < n; i++) {
    printf("address node %d = %p\n", n-i, (void *) cur);
    printf("value node %d = %d\n", n-i, cur->val);
    printf("reference node %d = %p\n", n-i, (void *) cur
        ->next);
    printf("-----------------------\n");
    if (cur) cur=cur->next;
}
```

Freeing the list

Free the list nodes (from the last node, i.e. the head to the first).

```
for (int i=0; i < n; i++) {
  struct node *cur = head;
  head  = cur->next;
  free(cur);
 }
```

Freeing the nodes is important to avoid memory leaks.

Other linked lists

A doubly-linked list is a list where each node has two references, one to the *previous* and one to the *next* node.

See C example 7.3 for an example of a doubly-linked list.

A circular list is a *simply-linked* list where the *last* node is connected to the *first*.

See C example 7.4 for an example of a circular list.

References

More about structures can be found in the online C manual or Chapter 6 The C Programming Langauge.

See also the code in C example 7.1 on Moodle.

More about linked lists can be found in Section 6.5 of The C Programming Langaugeor Section 10.2 of Cormen et al.'s Introduction to Algorithms.

See also the code in C example 7.2, C example 7.3, and C example 7.4 on Moodle.