COMP1511: Dynamic memory, Self Referential Structures - Linked List

Session 2, 2018

Dynamic memory allocation: `malloc`

- `malloc` allocates memory of a requested size (in bytes)
- Memory is allocated in "the heap", and it *lives* ***forever*** until we **free** it (or the program ends)
- *Important*: We MUST **free** memory allocated by `malloc`, should not rely on the operating system for cleanup.

```
malloc(number of bytes to allocate);
```

➔ returns a **pointer** to the block of allocated memory (i.e. the **address** of the memory, so we know how to find it!).

➔ **returns NULL** if insufficient memory available - you **must check** for this!

For example, let's assume we need a block of memory to hold 100,000 integers:

```
int *p = malloc( 100000 * sizeof(int) );
```

malloc : when it fails !

What happens if the allocation fails?

malloc returns NULL, and we need to check this:

```
int *p = malloc(1000 * sizeof(int));

if (p == NULL) {
    fprintf(stderr, "Error: couldn't allocate memory!\n");
    exit(1);
}
```

sizeof

- **sizeof** - C operator yields bytes needed for type or variable
- **sizeof (type)** or **sizeof variable**
- note unusual (badly designed) syntax - brackets indicate argument is a type
- use sizeof for every malloc call

```c
printf("%ld", sizeof (char));     // 1
printf("%ld", sizeof (int));      // 4 commonly
printf("%ld", sizeof (double));   // 8 commonly
printf("%ld", sizeof (int[10]));  // 40 commonly
printf("%ld", sizeof (int *));    // 4 or 8 commonly
printf("%ld", sizeof "hello");    // 6
```

free

- when we’re done with the memory allocated by **malloc** function, we need to release that memory using **free** function.
- For example,

```
int *p = malloc(1000 * sizeof(int));

if (p == NULL) {
    fprintf(stderr, "Error: couldn't allocate memory!\n");
    exit(1);
}

// do some thing here with the memory allocation
//

// free up the memory that was used
free(p);
```

`free`

- `free()` indicates you've finished using the block of memory
- Continuing to use memory after `free()` results in very nasty bugs.
- `free()` memory block twice also cause bad bugs.
- if program keeps calling `malloc()` without corresponding `free()` calls program's memory will grow steadily larger called a **memory leak**.
- Memory leaks major issue for long running programs.
- Operating system recovers memory when program exists.

Scope and Lifetime

- the variables inside a function only exist as long as the function does
- once your function returns, the variables inside are “gone”

What if we need something to “stick around” for longer?

Two options:

- make it in a “parent” function
- dynamically allocate memory

Lifetimes

Make it in a “parent” function,
for example:

```
void changeA(int *b, int size){
    b[2] = 55;
}
void main(void) {
    int a[10] = {0};
    changeA( a , 10);
    printf("%d", a[2] ); // prints 55
}
```

Allocate in a “parent” function

pass a pointer

Lifetimes

Dynamically allocate memory in a function and return a pointer,

For example:

```
int *getA(void){

    int *b = malloc(10 * sizeof(int));
    b[2] = 55;

    return b;
}

void main(void) {

    int *a = getA();

    printf("%d", a[2] ); // prints 55

    free(a);

}
```

Dynamically allocate in a function

return a pointer

free

Self-Referential Structures

We can define a structure containing
a pointer to the same type of structure:

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

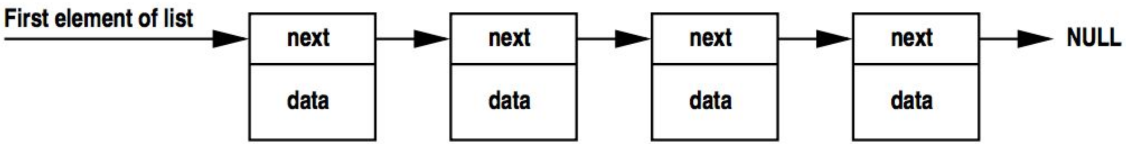
These "self-referential" pointers can be used to build larger "dynamic" data structures out of smaller building blocks.

Linked List

The most fundamental of these dynamic data structures is the *Linked List*:

- based on the idea of a sequence of data items or nodes
- linked lists are more flexible than arrays:
  - items don't have to be located next to each other in memory
  - items can easily be rearranged by altering pointers
  - the number of items can change dynamically
  - items can be added or removed in any order

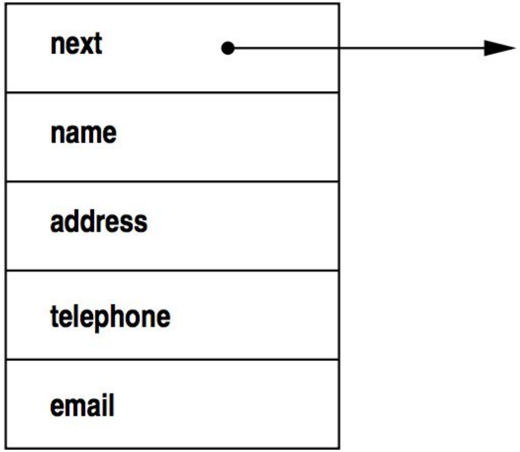
Linked List



- a *linked list* is a sequence of items
- each item contains data and a pointer to the next item
- need to separately store a pointer to the first item or "head" of the list
- the last item in the list is special
  it contains `NULL` in its `next` field instead of a pointer to an item

Example of List Item

Example of a list item used to store an address:



Example of List Item in C

```c
struct address_node {
    struct address_node *next;
    char *telephone;
    char *email;
    char *address;
    char *telephone;
    char *email;
};
```

List Items

List items may hold large amount of data or many fields.
For simplicity, we'll assume each list item need store only a single int.

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

List Operations

Basic list operations:

- create a new item with specified data
- search for a item with particular data
- insert a new item to the list
- remove a item from the list

Many other operations are possible.

Creating a List Item

```c
// Create a new struct node containing the specified dat
// and next fields, return a pointer to the new struct n

struct node *create_node(int data, struct node *next) {
    struct node *n;
    n = malloc(sizeof (struct node));
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    n->data = data;
    n->next = next;
    return n;
}
```

Building a list

Building a list containing the 4 ints: 13, 17, 42, 5

```c
struct node *head = create_node(5, NULL);
head = create_node(42, head);
head = create_node(17, head);
head = create_node(13, head);
```