Dynamic memory management

Dynamic memory

- Variables, constants are basic building blocks which are manipulated in a program to achieve the desired task.
- This means that you know what are all the entities you are going to work.
- That is, you know the size of the entities before hand.
- This is not completely true! Often you want to use a list of entities where the number of entities are unknown before hand.
- That is, you need to dynamically allocate memory.
- For example the array size may not be known before hand.

Dynamic memory

- Basically what we want is:
  - To allocate memory to hold entities.
  - To free the memory when the entities are no longer required.
  - We can access the entities by their addresses; we can not create a name for the dynamically allocated memory (since this requires declaration of names).
  - So, pointers are essential.

Dynamic memory allocation

- stdlib.h contains
  - void *malloc(size_t size);

    size_t is an alias for unsigned int.

    malloc returns a pointer to space for an object of the given size. It returns NULL if the request cannot be satisfied. The space is uninitialized.
  - void *calloc(size_t n, size_t size);

    returns a pointer to space for an array of n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero.
  - void free(void *p);

    releases the memory pointed by p; p must be a pointer to space allocated dynamically. It does nothing if p is NULL.

Dynamic memory allocation

- void *realloc(void *block, size_t size);

  realloc adjusts the amount of memory allocated to the block to size, copying the contents to the new location if necessary. block must be an allocated memory using dynamic memory functions.
- On success, the function returns the address of the reallocated block, which might be different from the address of the original block.
- On failure the function returns NULL.

examples

- int *p;
  p = (int *)malloc(sizeof(int));
  *p = 10;
  printf("%d", *p);
  free(p);
- the address returned by malloc needs to be type casted.

example

- ```
  int *p;
  p = (int *) calloc(10,  sizeof(int) );
  /* now p is the  starting address of an array of 10
     ints  */
  p[5] = 10;
  ```
  ```

Concepts of linked lists

- List is a set of items organized sequentially. An array is an example of list.
- In arrays sequential organization is provided by its index.
- One problem with arrays is that size of an array must be specified and which cannot be a variable. This might be not possible in many real life applications.
- One remedy is to use dynamic memory using malloc, calloc, etc.

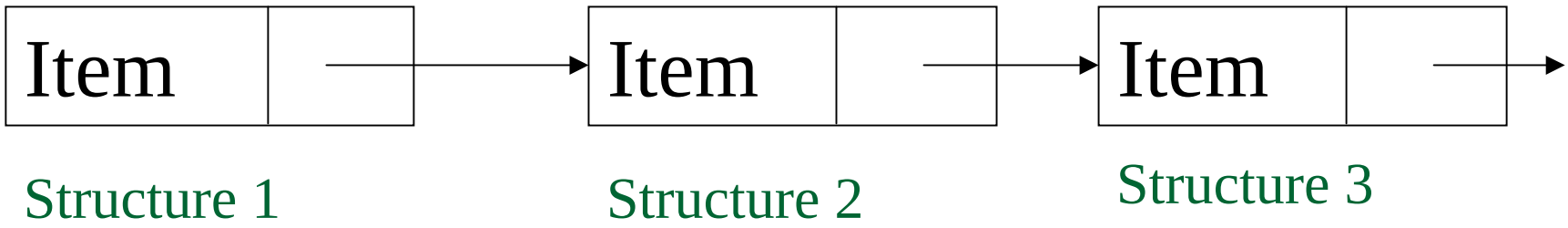
Arrays

- One more problem with arrays is: how to insert a new element in the middle of an array.
- Eg: int a[10] = { 0,1, 2, 3,5, 6,7};
- Now we want to make a[4] = 4, a[5] = 5, ….
- This can be achieved by several assignment statements. May be we can use a loop.
- This is an overhead if you want to modify the sequence by adding a new element in between.
- Similarly if you want to delete an element from the middle of an array. You need to pull forward all elements that follows the deleted element to fill the vacated space.

lists

- The problem with arrays is: it is not suitable for maintaining a list for which the size changes dynamically.
- Dynamic memory allocation cannot solve this problem.

Linked list




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

Linked list of names (in reverse order)

```
struct node {  char name[64];  struct node * next; };
main( )
{
        char s[64];
        struct node *head = NULL,  *new_node;
        do{
          puts("enter a name:");
          gets(s);
          if(strlen(s) > 0) {
                  new_node = (struct node *) malloc(sizeof(struct node));
                  strcpy(new_node->name,  s);
                  new_node -> next = head;
                  head = new_node;
                  }
        } while(strlen(s) > 0);
}
```

How to insert in between a linked list

- We want to insert a node s after the node pointed by where.
- void insert(struct node *where, struct node s)

```
{
        struct node *temp;
        temp = where ->next;
        where->next = &s;
        s.next = temp;
}
```

How to delete from middle

- The following fn deletes a node which is after the node pointed by p
- void delete(struct node *p)

```
{
        struct node *temp;
        temp = p ->next ->next;
        free(p->next);
        p->next = temp;
}
```

Searching a linked list

- Assume that structures contain some info like roll_no, marks, etc.
- struct node *search(struct node *head, int roll_no)

```
{
        struct node *temp;
        temp = head;
        while(temp != NULL) {
                if(temp->roll_no == roll_no)
                        return(temp);
                else temp = temp->next;
        }
        return NULL;
}
```

Other linked lists

- What we saw is also called as singly linked linear list.
- In contrast there could be doubly linked linear list,
- there could be circular linked list.

Don't we have any drawbacks of linked lists ?

- If the list is a static one, then the next pointers in the linked list consumes additional space (which is not consumed by arrays).
- Array is a randomly accessible data structure, whereas linked lists are sequential.
- Searching a sorted array might be very easy when compared with searching a sorted linked list.