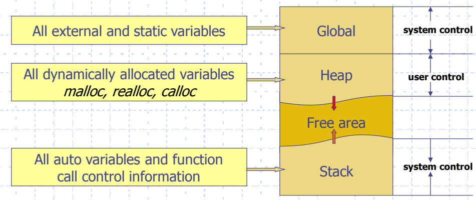
2. Linked Lists

- C review: Run time memory management and dynamic memory allocation in C.
- Linked lists: Structure and operations, comparison with arrays.
- Ordered linked lists and operations.
- Doubly linked lists and operations.

Runtime Memory Management:



Example:

- Suppose we want to design a program for handling student information:

```
typedef struct {
        char name[20];
        int grade;
} student;
```

- Question: how to create a table of student records?
  - a) static array: student stable[MAX_STUDENTS];
  - b) dynamic: Table? List? …

Dynamical Memory Allocation:

- C requires that the number of items in an array to be known at compile time. Too big or too small?
- Dynamical memory allocation allows us to specify an array's size at run time.
- Two important library functions are malloc(), which allocates space from HEAP, and free(), which returns the space allocated by malloc() back to HEAP for reuse.

Example:

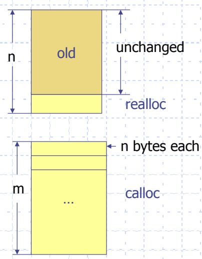
```
/* allocate and free an array of students, with error check */
#include <stddef.h> // including definition of NULL
#include <stdlib.h>  // including definition for malloc/free

student *table_create(int n){
   student* tp;
   if ((tp = malloc(n*sizeof(student))) != NULL)
         return tp;
   printf("table_create: dynamic allocation failed.\n");
   exit(0);
}

void table_free(student *tp){
   if (tp != NULL) free(tp);
}
```


Some Comments:

- Don't assume malloc() will always succeed.
- Don't assume the dynamically allocated memory is initialized to zero.
- Don't modify the pointer returned by malloc().
- free() only frees pointers obtained from malloc(). Don't access the memory after it has been freed.
- Don't forget to free memory which is no longer in use (garbage).

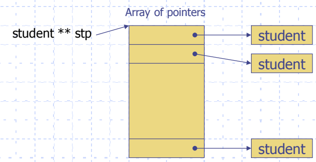
Allocate and free dynamic memory

- #include <stdlib.h>
- void* malloc(size_t n)
  allocates n bytes and returns a pointer to the allocated memory, the memory is not cleared
- void* realloc(void* p, size_t n)
  changes the size of the memory block pointed to by p to n bytes. The contents will be unchanged to the minimum of the old and new sizes.
- void* calloc(size_t m, size_t n)
  allocates memory for an array of m elements of n bytes each, and returns a pointer to the array.
- void free(void* p)
  frees the memory block pointed to by p.



Example: student table

- To create a student table dynamically.
  (1) do not know how many students,
  (2) keyboard input
- Requirement: the table holds exact number of pointers to student records;



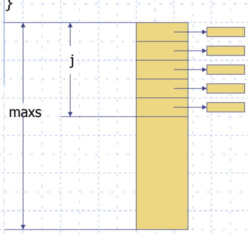

```
/* creates a student record dynamically and returns
   the pointer to the student record                */

student* make_student(char* name, int grade){
      student* sp;
      if ((sp = malloc(sizeof(student))) == NULL){
            printf("make_student: dynamic allocation failed.\n");
            exit(0);
      }
      strcpy(sp->name, name);
      sp->grade = grade;
      return sp;
}
```



```
#define CHUNK 5
student** make_table(int* num){
   int j = 0, maxs = CHUNK, grade;
   char name[20];
   student** stp;
   stp = (student**) malloc(maxs*sizeof(student*));
   while (2 == scanf("%s%d\n", name, &grade)){
         if (j >= maxs){
           maxs += CHUNK;
           stp = (student**) realloc(stp, maxs*sizeof(student*));
         }
         stp[j++] = make_student(name, grade);
   }
   if (j < maxs)
         stp = (student**) realloc(stp, j*sizeof(student*));
   *num = j;
   return stp;
}
```



```
int main(){
   student** cis2520;
   int num;
      cis2520 = make_table(&num);
      printf("The total number of students: %d\n", num);
      // other processing
}
```



j: counter of students
maxs: number of entries in the table

If the table is not big enough, Add another 5 entries;
Upon complete, if the table has unused entries, then return them.
Note: we only expand or truncate the pointer array (stp), student records are not changed.

Nested dynamic memory allocation:

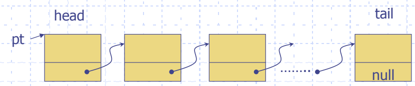
- A different structure:

```
typedef struct {
        char *name; // instead of char name[20]
        int grade;
} student;
```

```
student* make_student(char* name, int grade){
        student* sp;
        if ((sp = malloc(sizeof(student))) != NULL &&
          (sp->name = malloc(strlen(name) + 1))) != NULL){
                strcpy(sp->name, name);
                sp->grade = grade;
                return sp;
        }
        printf("make_student: dynamic allocation failed.\n");
        exit(0);
}

void free_student(student* sp){
        free(sp->name);             // must release name field first
        free(sp);
}
```


Linked list:

- A linked list represents a sequence:
  Every node but one has a predecessor, and every node but one has a successor.




```
/* recursive definition */
typedef struct node {
        int data; // whatever useful in the node
        struct node* next; // link to the next node
} node;
```


Example: student list

- data structure:

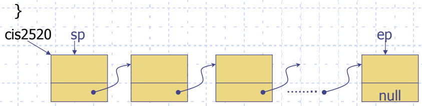
```
typedef struct student{
        char name[20];
        int grade;
        struct student* next;
} student;
```

- make_student() function not changed


```
student* make_list(int* num){
    int j = 0, grade;
    char name[20];
    student *sp = NULL, *ep = NULL;
    while (2 == scanf("%s%d\n", name, &grade)){
        if (sp == NULL)  // empty list
            sp = ep = make_student(name, grade);
        else {  // not empty,  insert at the end of the list
            ep->next = make_student(name, grade);
            ep = ep->next;
        }
        j++;
    }
    if (ep != NULL) ep->next = NULL; // last node of the list
    *num = j;
    return sp;
}
```



```
int main(){
    student* cis2520;
    int num;
        cis2520 = make_list(&num);
        printf("The total number of students: %d\n", num);
        // other processing
}
```



Other useful functions:

```
/* find the student record wrt name */
student* find(student* sp, char* name){
    while (sp && (strcmp(sp->name, name) != 0))
        sp = sp->next;
    return sp;
}

/* print student list */
void print_list(student* sp){
    while (sp){
        printf("Name: %s Grade: %d \n", sp->name, sp->grade);
        sp = sp->next;
    }
}
```

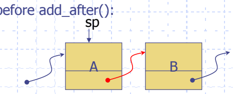
Recursive versions of print_list()

```
/* print student list recursively (from head to tail)*/
void print_list_1(student* sp){
    if (sp){
        printf("Name: %s Grade: %d \n", sp->name, sp->grade);
        printf_list_1(sp->next);
    }
}

/* print student list recursively (from tail back to head)*/
void print_list_2(student* sp){
    if (sp){
        printf_list_2(sp->next);
        printf("Name: %s Grade: %d \n", sp->name, sp->grade);
    }
}
```


Insertion

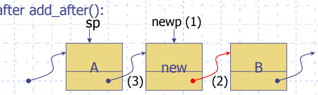
```
/* add a new student record after the node pointed by sp */
void add_after(student* sp, char name*, int grade){
    student* newp = make_student(name, grade); // (1)
    newp->next = sp->next;      // (2)
    sp->next = newp;            // (3)
}
```

before add_after():



Q: can we do add_before?

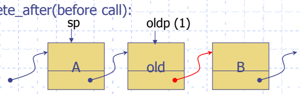
after add_after():



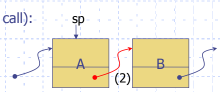
Deletion

```
/* delete the student record after the node pointed by sp */
void delete_after(student* sp){
    student* oldp;
    if (!sp || !sp->next) return;
    oldp = sp->next;             // (1)
    sp->next = oldp->next;       // (2)
    free(oldp);
}
```

delete_after(before call):



delete_after(after call):



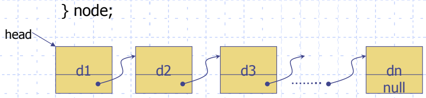
Linked list vs Array:

- Array:
  - static storage allocation
  - storage is contiguous
  - random access (using index)
  - insert and delete must shift existing data
- Linked List:
  - dynamic storage allocation
  - storage is not contiguous
  - sequential access only
  - insert and delete do not change existing data

Ordered Linked List:

- Insert nodes in their sorted position.

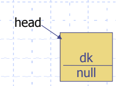
```
typedef struct node{
        int data;
        struct node* next;
} node;
```



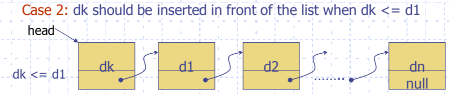
where d1 <= d2 <= d3 <= ... <= dn

- Question: how to insert dk into the list?

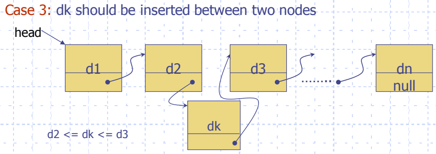
Case 1: head == NULL



Case 2: dk should be inserted in front of the list when dk <= d1



Case 3: dk should be inserted between two nodes



can we declare:

```
void insert(node * hd, int data){ … }
```

suppose we have the code:

```
node* head = NULL;
insert(head, 2);
insert(head, 5);
…
```

NO, because list head will be modified in both Case 1 and Case 2. That is, the content of head has to be changed.

```
void insert(node ** hp, int data){ … }

node* head = NULL;
insert(&head, 2);
insert(&head, 5);
…
```



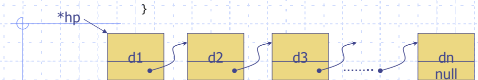
```
/* Version 1: insert a new node (data) into a list pointed to by *hp */

void insert(node** hp, int data){
    node* new, *prev == NULL, *curr;
    new = make_node(data);              // suppose we have this function
    curr = *hp;                         // get head pointer

    while (curr && data > curr->data){  // find position
        prev = curr;
        curr = curr->next;
    }
    if (prev == NULL){                  // or if (!prev)
        new->next = *hp;                // insert in front
        *hp = new;
    }
    else {                              // insert after prev
        prev->next = new;
        new->next = curr;
    }
}
```



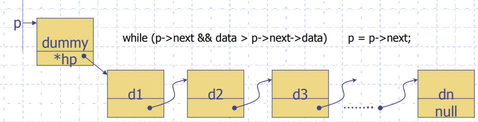
```
/* Version 2: insert a new node (data) into a list pointed to by *hp */

void insert(node** hp, int data){
    node dummy, *new, *p;

    new = make_node(data);                  // suppose we have this function
    p = &dummy;
    dummy.next = *hp;                       // set head in dummy

    while (p->next && data > p->next->data)     // find position
        p = p->next;

    new->next = p->next;                    // always insert after p
    p->next = new;
    *hp = dummy.next;
}
```


Version 1:

```
while (curr && data > curr->data){
        prev = curr;
        curr = curr->next;
}
```



Version 2:

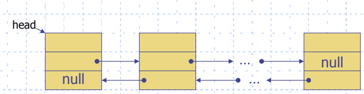



```
/* delete the node (data) from a list pointed to by *hp */
void delete(node** hp, int data){
        node dummy, *old, *p;
        p = &dummy;
        dummy.next = *hp;                           // set head in dummy
        while (p->next && data != p->next->data)    // find position
                p = p->next;
        if (p->next){
                old = p->next;                      // get the node to be deleted
                p->next = p->next->next;
                free(old);                          // free the node
        }
        *hp = dummy.next;
}
```


Doubly Linked List:

- A list which can be traversed either forward or backward:

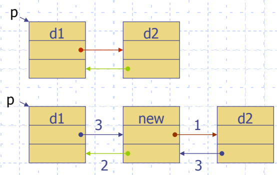
```
typedef struct node{
        int data;
        struct node* next;
        struct node* prev;
} node;
```



Insertion operation:

◆ insert_after the node pointed by p:

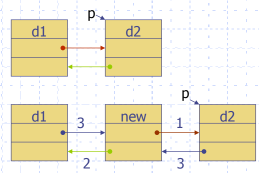
```
new->next = p->next;              // 1 (red link)
new->prev = p;                    // 2 (green link)
p->next = p->next->prev = new;    // 3 (blue links)
```



Insertion operation:

◆ insert_before the node pointed by p:

```
new->next = p;                    // 1 (red link)
new->prev = p->prev;              // 2 (green link)
p->prev = p->prev->next = new;    // 3 (blue links)
```



Deletion operation:

◆ delete the node pointed by p:

```
p->next->prev = p->prev;    // 1 (green link)
p->prev->next = p->next;    // 2 (blue link)
```