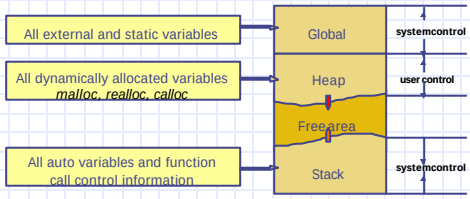
Runtime Memory Allocation:



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 the number of items in an array to be known at compile time. Too big or too small?
- Dynamical memory allocation allow 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 later.

Example:

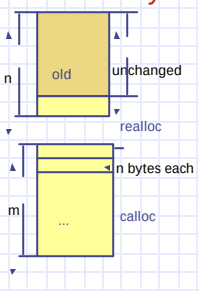
```
/* allocate and free an array of students, with error check */
#include <stddef.h> // definition of NULL
#include <stdlib.h>  // definition for malloc/free

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 pointers obtained from **malloc**, and don't access the memory after it has been freed.
- Don't forget to **free** memory which is no longer in use (garbage).

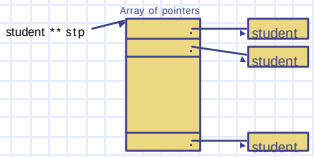
Allocate and free dynamic memory

- `#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: 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;
}
```

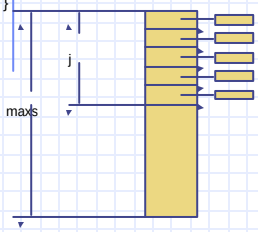
```
/* copy s to t, suppose t long enough*/
void my_strcpy(char* t, char* s){ while (*t++ = *s++); }
```



```
#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:

- suppose :

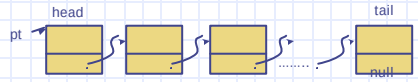
```
typedef struct {
        char *name; // instead of char name[20]
        int grade;
} student;
```



```
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:

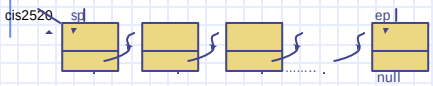
```
typedef struct student{
        char name[20];
        int grade;
        struct student* next;
} student;
```

- 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)
            sp = ep = make_student(name, grade);
        else {
            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:

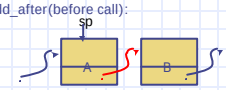
```
/* find the student record wrt name */
student* find(student* sp, char* name){
    while (sp && (strcmp(sp->name, name) != 0))
        sp = sp->next;
    return sp;
}

/* 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);
    }
}
```

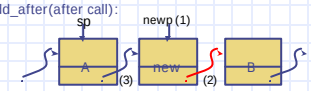
```
/* add a new student record after sp */
void add_after(student* sp, char name*, int grade){
    student* newp = make_student(name, grade); // (1)
    newp->next = sp->next;      // (2)
    sp->next = newp;            // (3)
}
```

add_after(before call):



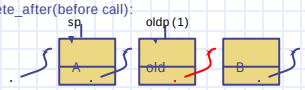
Q: can we do add_before?

add_after(after call):

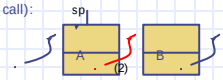



```
/* delete the student record after sp */
void delete_after(student* sp){
    student* oldp;
    if (!sp || !sp->next) return;
    oldp = sp->next;            // (1)
    sp->next = oldp->next;      // (2)
    free(oldp);
}
```

delete_after(before call):



delete_after(after call):



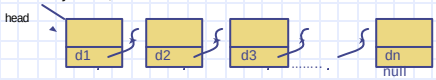
Linked list vs Array:

- Array:
  - static storage allocation
  - storage is contiguous
  - random access (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;
```

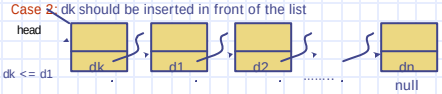


where d1 <= d2 <= d3 <= ... <= dn

- Question: how to insert dk into the list?

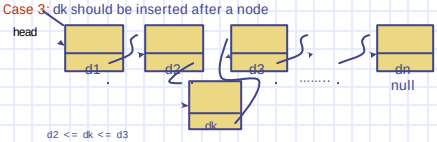
Case 1: head == NULL

Case 2: dk should be inserted in front of the list



dk <= d1

Case 3: dk should be inserted after a node



d2 <= dk <= d3

- 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.

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

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

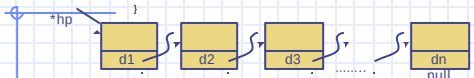
```
/* Version 1: insert a new node (data) into a list pointed to by *hp */

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;
    }
}
```



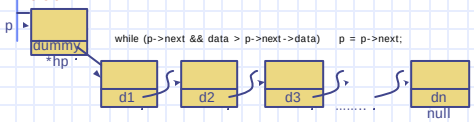
```
/* Version 2: insert a new node (data) into a list pointed to by *hp */

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:



Suppose:


```
node *head, *p, *q, *temp;
int data;

p = head;
q = NULL;
```

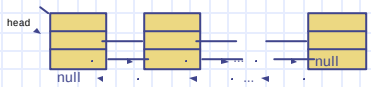


```
/* delete the node (data) from a list pointed to by *hp */
void delete(node** hp, int data){
    node dummy, *old, *p;
    p = &dummy;
    dummy.next = *hp;                       // set head in dummy
    while (p->next && data != p->next->data)    // find position
        p = p->next;
    if (p->next){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

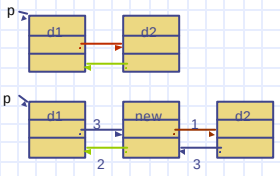
```
typedef struct node{
        int data;
        struct node* next;
        struct node* prev;
} node;
```



Insert operation:

- insert_after:

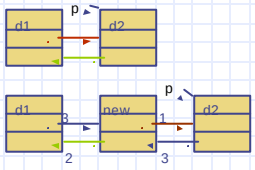
```
new->next = p->next;              // 1 (red link)
new->prev = p;                    // 2 (green link)
p->next = p->next->prev = new;    // 3 (blue links)
```



Insert operation:

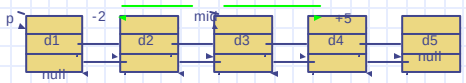
- insert_before:

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



Example:

- Suppose we have a doubly linked list:
  (1) find the mid node,
  (2) along mid's prev link, decrease each node value by 2,
  along mid's next link (include mid node), increase each node value by 5.




```
#define NEXT 0
#define PREV 1
typedef  struct node{
    int data;
    struct node *link[2];
} node;

void traverse(node* p, int dir, void (*fp)(node*)){
    while (p){
        (*fp)(p);          // call function fp with argument p
        p = p->link[dir];
    }
}

void dec2(node* p){ p->data -= 2; }

void inc5(node* p){ p->data += 5; }
```



```
void processing(node* p){
    node* mid = p;

    while (p && p->link[NEXT] && p->link[NEXT]->link[NEXT]){
        mid = mid->link[NEXT];
        p = p->link[NEXT]->link[NEXT];
    }
    traverse(mid, PREV, dec2);      // any problem here?
    traverse(mid, NEXT, inc5);
}
```

Function Pointer:

*type (*fp)(parameter_list);*

Function name is the pointer to that function.

Generic Functions:

- A generic function is one that can work on any underlying C data type.
- Generic functions allow us to reuse programs by adding a small piece of type-specific code.
- For example, standard C library provides two generic functions: bsearch searches an arbitrary array, and qsort sorts an arbitrary array.

Review of Pointers to Functions:

- A pointer to a function contains the address of the function in memory. Similar to an array name which is the starting address of the first array element, a function name is the starting address of the function code.
- Pointers to functions can be passed to functions, returned from functions, stored in an array, and assigned to other function pointers.

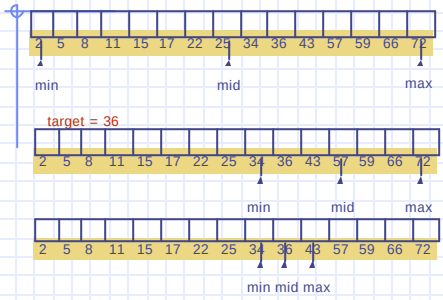
Pointers to Functions:

```
int compare(int a, int b){
        // implementation
}

int main(){
        int cr1, cr2;
        int (*pf)(int, int); // declare a pointer to function
        pf = compare;        // assign a function pointer to pf
        cr1 = (*pf)(2, 3);   // call the function
        cr2 = pf(2, 3);      // same as above but not recommended
}
```


Binary search function:

- binary search is used to find a target value in a sorted table.
- we start by comparing the target value with the table's middle element.
- since the table is sorted, so if the target is larger, we can ignore all values smaller than the middle element, and *vice versa.*
- We stop when we've found the target or no values left to search.

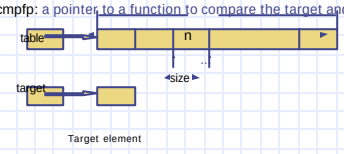

```
int* bsearch(int target, int *table, int n){
    int *min = table;
    int *max = table + (n – 1);
    int *mid;
    int k = n/2;

    while (min < max) {
            mid = min + k;
            if (target == *mid)  return mid;
            else if (target > *mid)
                    min = mid + 1;
            else
                    max = mid - 1;
            k /= 2;
    }
    return NULL;
}
```

Generic Binary Search:

```
void * bserach(const void *target, const void *table, size_t n,
        size_t size, int (*cmpfp)(const void *, const void*));
```

target: a pointer to the element we are searching for
table: the array we are searching
n: the size of the array
size: the size of an array element
cmpfp: a pointer to a function to compare the target and an element



bsearch Function:

```
void * bserach(const void *target, const void *table, size_t n,
        size_t size, int (*cmpfp)(const void *, const void*)){
    void *min = table, *max = table + (n – 1)*size;
    void *mid;
    int k = n/2;
    int cr;
    while (min < max) {
        mid = min + k*size;
        cr = (*cmpfp)(target, mid) ;
        if (cr == 0) return mid;
        else if (cr > 0)        min = mid + size;
        else                    max = mid - size;
        k /= 2;
    }
    return NULL;
}
```


Invoking bsearch:

- bsearch returns a pointer to the matching array element if it finds one, or NULL otherwise.
- Suppose we have an int table intTable and a string table stringTable, we might call:

```
int key = 78;
int* ip = bsearch(&key, intTable, ISIZE, sizeof(int), cmpInt);
char* sp = bsearch("Tom Wilson", stringTable, SSIZE,
                   sizeof (stringTable[0]), cmpStr);
```


Type-specific Functions:

```
int cmpInt(const void *tp, const void *ep){
    int it = *(int*) tp; // get target value
    int ie = *(int*) ep; // get array element value
    return (it == ie) ? 0 : ((it < ie) ? –1 : 1);
}

int cmpStr(const void *tp, const void *ep){
    char *ctp = (char*) tp; // casting target to char pointer
    char *cep = *(char**) ep; // get array element value (a char pointer)
    return strcmp(ctp, cep);
}
```


- (1) Write a comparison function:

```
int cmpStudent(const void* pt, const void *ps);
```

which calls strcmp to compare two student names.

- (2) suppose a new student and a pointer variable are defined as follows:

```
struct student s = {"John Smith", 21};
struct student *pp;
```

Write the call statement to bsearch to find if s is in stable and assign the returned value to pp.