

## 6.087 Lecture 6 – January 19, 2010

---

- Review
- User defined datatype
  - Structures
  - Unions
  - Bitfields
- Data structure
  - Memory allocation
  - Linked lists
  - Binary trees

# Review: pointers

---

- Pointers: memory address of variables
- '&' (address of) operator.
- Declaring: `int x=10; int * px= &x;`
- Dereferencing: `*px=20;`
- Pointer arithmetic:
  - `sizeof()`
  - incrementing/decrementing
  - absolute value after operation depends on pointer datatype.

# Review: string.h

---

- String copy: `strcpy()`, `strncpy()`
- Comparison: `strcmp()`, `strncmp()`
- Length: `strlen()`
- Concatenation: `strcat()`
- Search: `strchr()`, `strstr()`

# Searching and sorting

---

## Searching

- Linear search:  $O(n)$
- Binary search:  $O(\log n)$ . The array has to be sorted first.

## Sorting

- Insertion sort:  $O(n^2)$
- Quick sort:  $O(n \log n)$

## 6.087 Lecture 6 – January 19, 2010

---

- Review
- User defined datatype
  - Structures
  - Unions
  - Bitfields
- Data structure
  - Memory allocation
  - Linked lists
  - Binary trees

# Structure

---

Definition: A structure is a **collection of related variables** (of **possibly different types**) grouped together under a single name. This is an example of **composition**—building complex structures out of simple ones.

Examples:

```
struct point
{
    int x;
    int y;
};
```

*/\* notice the ; at the end \*/*

```
struct employee
{
    char fname[100];
    char lname[100];
    int age;
};
```

*/\* members of different type \*/*

# Structure

---

- **struct** defines a new datatype.
- The name of the structure is optional.  
**struct** {...} x,y,z;
- The variables declared within a structure are called its *members*
- Variables can be declared like any other built in data-type.  
**struct** point ptA;
- Initialization is done by specifying values of every member.  
**struct** point ptA={10,20};
- Assignment operator copies every member of the structure (be careful with pointers).

# Structure (cont.)

---

More examples:

```
struct triangle
```

```
{
```

```
    struct point ptA;
```

```
    struct point ptB;
```

```
    struct point ptC;
```

```
};
```

```
/*members can be structures*/ self referential*/
```

```
struct chain_element
```

```
{
```

```
    int data;
```

```
    struct chain_element* next
```

```
};
```

```
/*members can be
```

```
self referential*/
```



## Structure (cont.)

---

- Individual members can be accessed using '.' operator.  
`struct` point pt={10,20}; `int` x=pt.x; `int` y=pt.y;
- If structure is nested, multiple '.' are required

```
struct rectangle  
{  
    struct point tl; /* top left */  
    struct point br; /* bot right */  
};  
struct rectangle rect;  
int tlx=rect.tl.x; /* nested */  
int tly=rect.tl.y;
```

# Structure pointers

---

- Structures are copied element wise.
- For large structures it is more efficient to pass pointers.  
`void foo(struct point * pp); struct point pt; foo(&pt)`
- Members can be accessed from structure pointers using `'->'` operator.

```
struct point p={10,20};  
struct point* pp=&p;  
pp->x = 10; /*changes p.x*/  
int y= pp->y; /*same as y=p.y*/
```

Other ways to access structure members?

```
struct point p={10,20};  
struct point* pp=&p;  
(*pp).x = 10; /*changes p.x*/  
int y= (*pp).y; /*same as y=p.y*/
```

why is the `()` required?

# Arrays of structures

---

- Declaring arrays of int: **int** x[10];
- Declaring arrays of structure: **struct** point p[10];
- Initializing arrays of int: **int** x[4]={0,20,10,2};
- Initializing arrays of structure:  
    **struct** point p[3]={0,1,10,20,30,12};  
    **struct** point p [3]={{0,1},{10,20},{30,12}};

# Size of structures

---

- The size of a structure is greater than or equal to the sum of the sizes of its members.
- Alignment

```
struct {  
    char c;  
    /*padding*/  
    int i;
```

- Why is this an important issue? libraries, precompiled files, SIMD instructions.
- Members can be explicitly aligned using **compiler** extensions.

```
__attribute__((aligned(x))) /*gcc*/  
__declspec(aligned(x)) /*MSVC*/
```

# Union

---

A union is a variable that may hold objects of different types/sizes in the same memory location. Example:

```
union data
{
    int idata;
    float fdata;
    char* sdata;
} d1,d2,d3;
d1.idata=10;
d1.fdata=3.14F;
d1.sdata="hello world";
```

## Unions (cont.)

---

- The size of the union variable is equal to the size of its largest element.
- **Important:** The compiler does not test if the data is being read in the correct format.

```
union data d; d.idata=10; float f=d.fdata; /* will give junk*/
```

- A common solution is to maintain a separate variable.

```
enum dtype{INT,FLOAT,CHAR};  
struct variant  
{  
    union data d;  
    enum dtype t;  
};
```

# Bit fields

Definition: A bit-field is a set of adjacent bits within a single 'word'. Example:

```
struct flag{  
    unsigned int is_color:1;  
    unsigned int has_sound:1;  
    unsigned int is_ntsc:1;  
};
```

- the number after the colons specifies the width in bits.
- each variables should be declared as **unsigned int**

## Bit fields vs. masks

CLR=0x1,SND=0x2,NTSC=0x4;	<b>struct</b> flag f;
x = CLR; x =SND; x =NTSC	f.has_sound=1;f.is_color=1;
x&= ~CLR; x&=~SND;	f.has_sound=0;f.is_color=0;
<b>if</b> (x & CLR    x& NTSC)	<b>if</b> (f.is_color    f.has_sound)

## 6.087 Lecture 6 – January 19, 2010

---

- Review
- User defined datatype
  - Structures
  - Unions
  - Bitfields
- Data structure
  - Memory allocation
  - Linked lists
  - Binary trees



# Digression: dynamic memory allocation

---

**void\*** malloc(size\_t n)

- malloc() allocates blocks of memory
- returns a pointer to **uninitialized** block of memory on success
- returns NULL on failure.
- the returned value should be cast to appropriate type using (). **int\*** ip=(**int\***)malloc(**sizeof(int)\*100**)

**void\*** calloc(size\_t n, size\_t size)

- allocates an array of n elements each of which is 'size' bytes.
- **initializes memory to 0**

**void** free(**void\***)

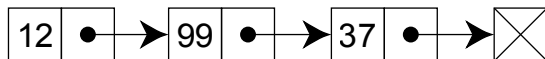
- Frees memory allocated by malloc()
- Common error: accessing memory after calling free

# Linked list

---

Definition: A dynamic data structure that consists of a sequence of records where each element contains a **link** to the next record in the sequence.

- Linked lists can be *singly linked*, *doubly linked* or *circular*. For now, we will focus on *singly linked* list.
- Every node has a *payload* and a link to the next node in the list.
- The *start (head)* of the list is maintained in a separate variable.
- End of the list is indicated by **NULL (sentinel)**.



# Linked list

---

```
struct node
{
    int data; /* payload */
    struct node* next;
};
struct node* head; /* beginning */
```

Linked list vs. arrays

	linked-list	array
size	dynamic	fixed
indexing	$O(n)$	$O(1)$
inserting	$O(1)$	$O(n)$
deleting	$O(1)$	$O(n)$

# Linked list

---

Creating new element:

```
struct node* nalloc(int data)
{
    struct node* p=(struct node*) malloc(sizeof(node));
    if (p!=NULL)
    {
        p->data=data;
        p->next=NULL;
    }
    return p;
}
```

# Linked list

---

Adding elements to front:

```
struct node* addfront(struct node* head,int data)
{
    struct node* p=nalloc(data);
    if (p==NULL) return head;
    p->next=head;
    return p;
}
```

# Linked list

---

Iterating:

```
for (p=head; p!=NULL; p=p->next)  
    /*do something*/
```

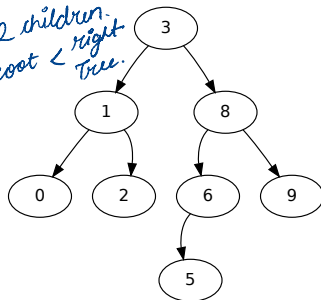
```
for (p=head; p->next!=NULL; p=p->next)  
    /*do something*/
```

# Binary trees

- A binary tree is a **dynamic data structure** where each node has at most two children. A binary **search tree** is a binary tree with **ordering among its children**.
- Usually, all elements in the left subtree are assumed to be "less" than the root element and all elements in the right subtree are assumed to be "greater" than the root element.

Binary Tree : atmost 2 children (Unsorted)

Binary Search Tree : atmost 2 children.  
left Tree < root < right Tree.



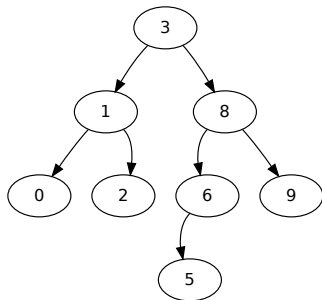
## Binary tree (cont.)

```
struct tnode
{
    int data; /*payload*/
    struct tnode* left;
    struct tnode* right;
};
```

The operation on trees can be framed as recursive operations.

### Traversal (printing, searching):

- **pre-order**: root, left subtree, right subtree
- **Inorder**: left subtree, root, right subtree
- **post-order**: right subtree, left subtree, root





## Binary tree (cont.)

---

### Add node:

```
struct tnode* addnode(struct tnode* root, int data)
{
    struct tnode* p=NULL;
    /* termination condition */
    if (root==NULL)
    {
        /* allocate node */
        /* return new root */
    }
    /* recursive call */
    else if (data < root->data)
        root->left=addnode(root->left, data)
    else
        root->right=addnode(root->right, data)
}
```

MIT OpenCourseWare

<http://ocw.mit.edu>

## 6.087 Practical Programming in C

January (IAP) 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>