Outline

Unions (continued)

Passing structures as function arguments

Returning structures from the functions

Linked list

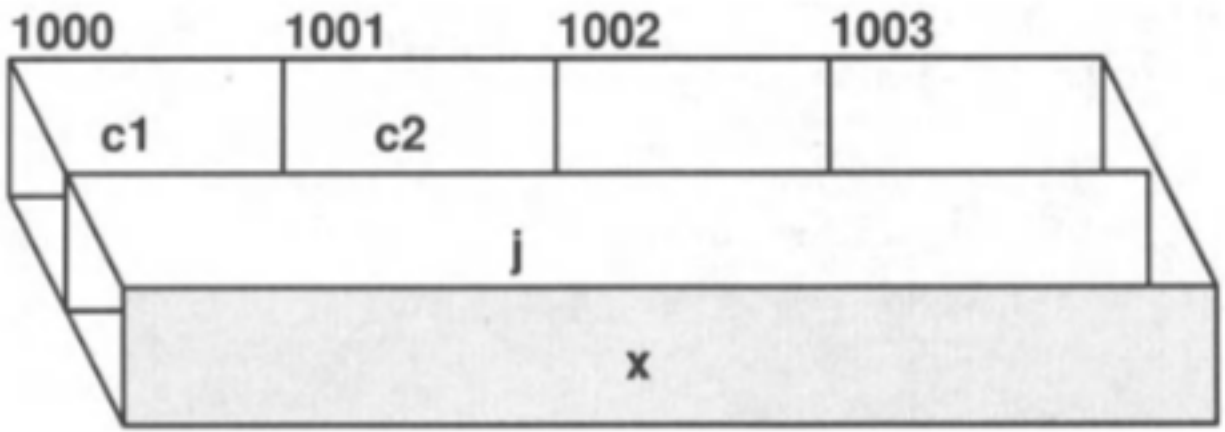
Unions (continued)

Unions (reminder)

- Unions are similar to structures except that the members are overlaid one on top of another, so members share the same memory.
- There are two basic applications for unions:
  - Interpreting the same memory in different ways.
  - Creating flexible structures that can hold different types of data.
- Example:

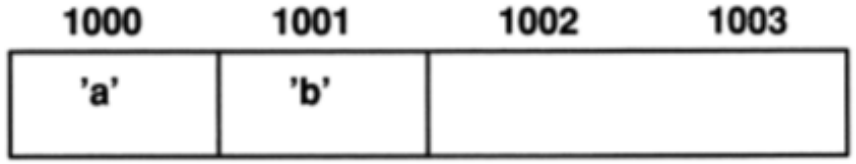
```
typedef union
{
   struct
   {
      char c1, c2;
   } s;
   long j;
   float x;
} U;

U example;
```



- Usage:

```
example.s.c1 = 'a';
example.s.c2 = 'b';
```



* If you make the assignment:
example.j = 5; // it overwrites the 2 chars, using all 4 bytes to store value 5.

Real life example for Unions in Structures

- Consider our PERSONALSTAT example (name, tcno, birth_date), we want to add additional information as follows:
  - Are you T.C. citizen?
  - If you are a T.C. citizen, in which city were you born?
  - If not a T.C. citizen, what is your nationality?

```
typedef struct {
   unsigned int day : 5;
   unsigned int month : 3;
   unsigned int year : 11;
} DATE;

typedef struct {
   char ps_name[20], ps_tcno[11];
   DATE ps_birth_date;
  // Bit field for TC citizenship:
   unsigned int TCcitizen : 1;
   char nationality[20];
   char city_of_birth[20];
} PERSONALSTAT;
```

Real life example for Unions in Structures

- Consider our PERSONALSTAT example (name, tcno, birth_date), we want to add additional information as follows:
  - Are you T.C. citizen?
  - If you are a T.C. citizen, in which city were you born?
  - If not a T.C. citizen, what is your nationality?

```
typedef struct {
   unsigned int day : 5;
   unsigned int month : 3;
   unsigned int year : 11;
} DATE;

typedef struct {
   char ps_name[20], ps_tcno[11];
   DATE ps_birth_date;
  // Bit field for TC citizenship:
   unsigned int TCcitizen : 1;
   char nationality[20];
   char city_of_birth[20];
} PERSONALSTAT;
```

```
typedef struct {
   unsigned int day : 5;
   unsigned int month : 3;
   unsigned int year : 11;
} DATE;

typedef struct {
   char ps_name[20], ps_tcno[11];
   DATE ps_birth_date;
  // Bit field for TC citizenship:
   unsigned int TCcitizen : 1;
   union{
      char nationality[20];
      char city_of_birth[20]
   } location;
} PERSONALSTAT;
```

Passing structures as function arguments

- There are two ways to pass structures as arguments:
  - **Pass the structure itself (pass by value):**

    PERSONALSTAT ps;

    …

    func(ps); // Pass by value. Passes an entire copy of the structure
  - **Pass a pointer to the structure (pass by reference):**

    …

    func(&ps); // Pass by reference. Passes the address of the structure

- Passing the address of a structure is usually faster because only a single pointer is copied to the argument area.
- Passing by value, on the other hand, requires that the entire structure be copied.
- There are only two circumstances when you should pass a structure by value:
  - The structure is very small (i.e., approximately the same size as a pointer).
  - You want to guarantee that the called function does not change the structure being passed. When an argument is passed by value, the compiler generates a copy of the argument for the called funct. The called function can only change the value of the copy

Passing structures as function arguments -2

- Depending on which method you choose, you need to declare the argument on the receiving side as either a structure or a pointer to a structure:
  - *func (PERSONALSTAT ps)* // Pass by value - the argument is a structure
  - *func (PERSONALSTAT* ptrps)* // Pass by reference - the argument is a pointer to a structure.

- **Note that** the argument-passing method you choose determines which operator you should use in the function body:
  - the dot operator if a structure is passed by value
  - the right-arrow operator if the structure is passed by reference.

Returning structures from the functions

- Just as it is possible to pass a structure or a pointer to a structure, it is also possible to return a structure or a pointer to a structure.

- As with passing structures, you generally want to return pointers to structures because it is more efficient.

- // Define a function that returns a struct:

```
struct tagname func1  (struct tagname st){
    …
    return st;  // Return an entire struct
```

- // Define a function that returns a pointer to a struct:

```
struct tagname * func2 (){
    static struct tagname pst;
    return &pst;  // Return the address of a struct
```

- Note, however, that if you return a pointer to a structure, the structure must have fixed duration. Otherwise, it may not be valid once the function returns.

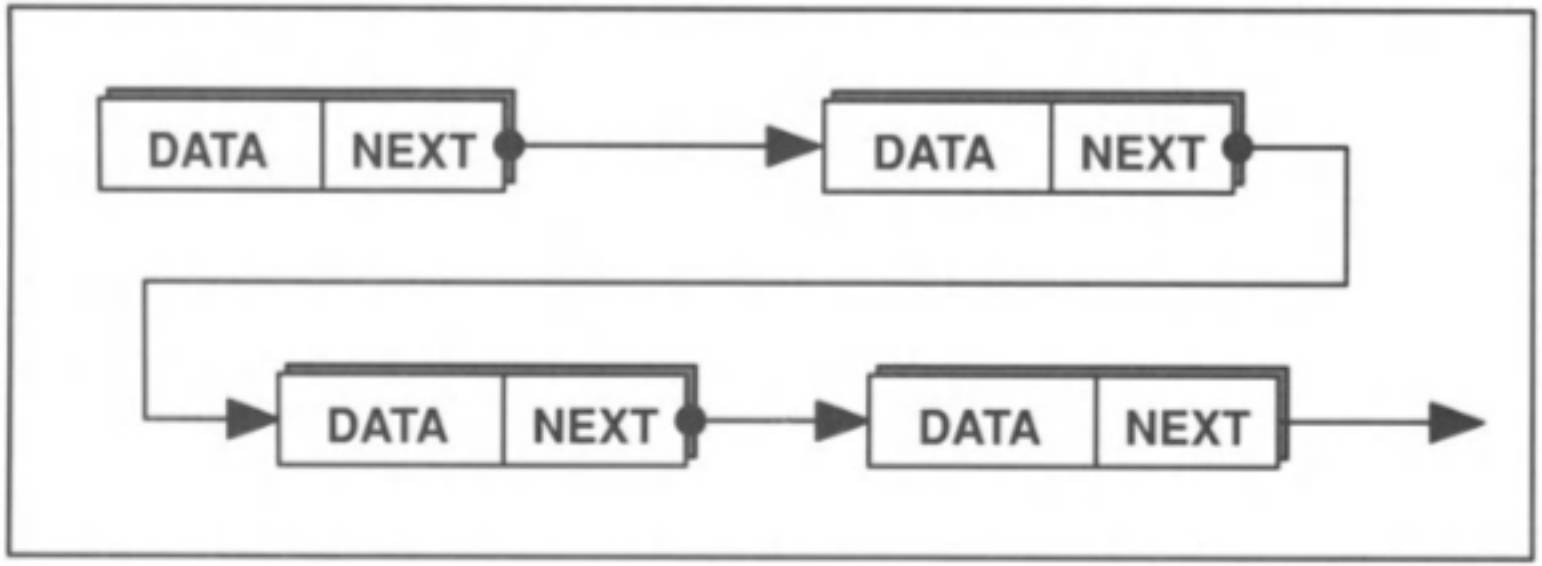
Trigonometric functions example with structures

Linked list

- We have used an **array of structures** to handle groups of data.
- This is OK, when you know beforehand exactly how many structures you will have.
- When the number is unknown, arrays can be extremely costly since:
  - They force you to allocate enough memory for the worst-case situation.
  - This memory is useless if you use only a fraction of the array elements.
  - Moreover, if you need to access more memory than you initially allocated, your program will fail.
- The obvious solution is to be able to allocate memory for new structures as needed, through the runtime library routines malloc(), calloc(), realloc():
  - But successive calls to these routines will not guarantee that the structures will be placed contiguously in memory.

- So, we need a technique for connecting all the structures together.
- The most common way to do this is through a construct called a **linked list**.
- A linked list is a **chain of structures** that are linked one to another, like sausages.
- In the linked-list scheme, each structure contains an extra member, which is a pointer to the next structure in the list.

Linked list



In a linked-list application, you need to perform the following operations:

- Create a list element
- Add elements to the end of a list
- Insert elements in the middle of a list
- Remove an element from a list
- Find a particular element in a list

Creating a Linked-List Element

- To make the function as general as possible, we use the name *ELEMENT*, which gives no clue about the actual type of data being manipulated:

```
ELEMENT *create_list_element()
{
  ELEMENT *p;

  p = (ELEMENT *) malloc( sizeof( ELEMENT ) );
  if (p == NULL)
  {
    printf( "create_list_element: malloc failed.\n");
    exit( 1 );
  }
  p->next = NULL;
  return p;
}
```

// ELEMENT becomes synonymous with struct personalstat (see the example code):

typedef struct personalstat ELEMENT;

Adding Elements to the Linked List

- The create_list_element() function allocates memory, but it doesn't link the element to the list.
- For this, we need an additional function, add_element():

```
static ELEMENT *head; // serves as a pointer to the beginning of the linked list
void add_element(ELEMENT *e){
    ELEMENT *p;
    // if the 1st element (the head) has not been created, create it now:
    if(head==NULL){
            head=e;             return;
    }
    // otherwise, find the last element in the list:
    // Span through each element testing to see whether p.next is NULL.
    // If not NULL, p.next must point to another element.
    // If NULL, we have found the end of the list , end for loop.
    for (p=head; p->next != NULL; p=p->next); // null statement
    // append a new structure to the end of the list
    p->next=e;
}
```

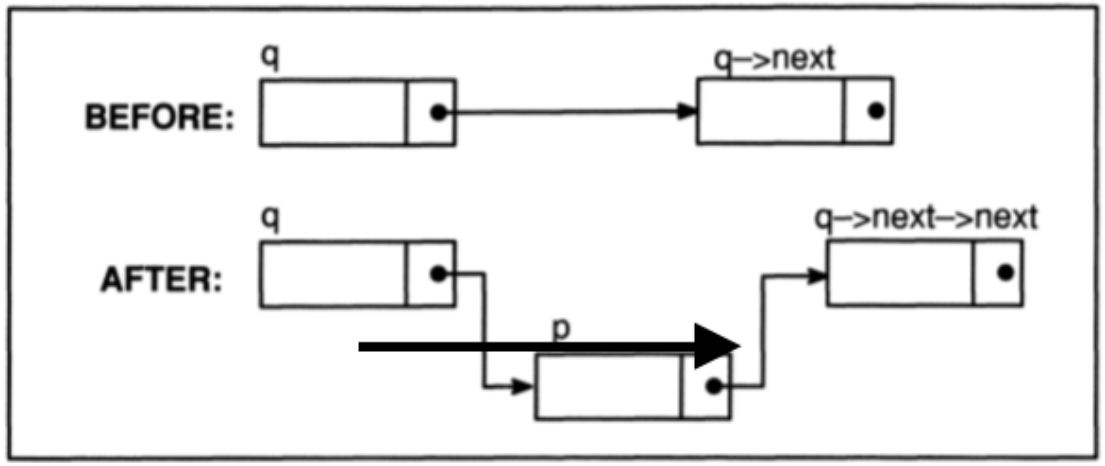
Create a linked list containing 10 *personalstat* structures:

```
static ELEMENT *head;
main(){
   for(int j=0; j<10; j++)
        add_element( create_list_element());
}
```

Inserting an Element to the Linked List

- To insert an element in a linked list, you must specify where you want the new element inserted.
- Insert function accepts 2 pointer arguments, p and q, and inserts the structure pointed by p, just after the structure pointed by q.

```
void insert_after(ELEMENT *p, ELEMENT *q){
    // if p and q are same or NULL, or if p already follows q, report that:
    if(p==NULL || q==NULL || p==q || q->next == p){
            printf("insert_after(): Bad arguments \n");
            return;
    }
    p->next=q->next;
    q->next=p;
}
```



Deleting an Element from the Linked List

- To delete an element in a linked list, you need to find the element before the one you are deleting so that you can bond the list back together after removing one of the links.
- You also need to use the free() func, to free up the memory used by the deleted element.

```
void delete_element(ELEMENT *goner){
    ELEMENT *p
    if(goner == head)
            head=goner->next;
    else // find element preceding the one to be deleted:
            for(p=head; (p!=NULL) && (p->next != goner); p=p->next); // null statement

    if(p == NULL){
            printf("delete_element(): could not find the element \n");
            return;
    }
    p->next=p->next->next;
    free(goner);
}
```

Finding an Element in the Linked List

- There is no easy way to create a general-purpose find() function because you usually search for an element based on one of its data fields (e.g. person's name), which depends on the structure being used.
- To write a general-purpose find() function, you can use function pointers (remember earlier classes!!)
- The following function, based on the personalstat structure, searches for an element, whose *ps_name* field matches with the given argument

```
ELEMENT *find( char * name){
    ELEMENT *p;
    for(p=head; p!= NULL; p=p->next)
            if(strcmp(p->ps_name, name) == 0) // strcmp() returns 0, if 2 strings are same
                    return p;
    return NULL;
}
```

- Check the example code shown in the class!!