Announcements

1. Everything through today is now on the Canvas and nob web sites
2. On Wednesday, May 10, we will resume in-person classes
3. I will also hold office hours in person beginning then
4. The midterm will be Friday, May 12
Structures

• Data structure used to group elements of a different type together
• Example: student registration number database
  • See element below

```c
struct student {
    char *name; /* student name */
    int regnumber; /* registration number */
};
```
Referring to a Structure

Here’s how you declare a variable of the structure:

```c
struct student xyzzy, *pxyzzy;
```

It’s clumsy to write that, so you can define an alias for the type:

```c
typedef struct student STUDENT;
```

The latter essentially produces a new type, STUDENT, that can be used wherever struct student can:

```c
STUDENT xyzzy, *pxyzzy;
```
Another Declarations

struct student {
    char *name;    /* student name */
    int regnumber; /* registration number */
} xyzzy, *pxyzzy;

• Declares type struct student and 2 variables, xyzzy (an instance of struct student) and pxyzzy (a pointer to an instance of struct student)
And Now, With a Typedef

typedef struct student {
    char *name; /* student name */
    int regnumber; /* registration number */
} STUDENT;

STUDENT xyzzy, *pxyzzy;

This defines a new type, STUDENT, which is the same as the type struct student. Here xyzzy is a variable of type STUDENT and pxyzzy is a pointer to an instance of STUDENT.
But Be Careful

• `typedef` defines an alias for a type
• `#define` does textual substitution

```c
typedef int *PINT;
PINT a, b, c
```

• Now `a`, `b`, and `c` are all pointers to integers

```c
#define PINT int *
PINT a, b, c;  /* becomes int * a, b, c; */
```

• Now `a` is a pointer to an integer, and `b` and `c` are integers
Linked List

• A list composed of instantiations of structures
  • One element is whatever is to be sorted (int, for us)
  • Another element is a pointer to the next element; NULL if none
Structure for This List

```c
struct node {
    int num;
    struct node *next;
};
struct node *list;
```

This holds the integer that you read in

This holds the pointer to the next element in the linked list; it’s NULL if it’s at the end

This points to the first element of the list
Changing How Memory Is Allocated

• Now you can allocate memory one element (“node”) at a time
• Insertion at beginning is like this (see “linked.c”, ll. 72–76):
  • new->next = first;
  • list = new;
• Insertion in the middle between prev and succ is (see “linked.c”, ll. 78–97):
  • new->next = succ;
  • prev->next = new;
• Insertion at the end nomore of the list (same as above):
  • nomore->next = new;
Insertion

headList

1

5

12

23

49
Insertion: At the Beginning of the List

First, change the pointer in the new node to point to the head of the list (where headList points; just copy the pointer)
Insertion: At the Beginning of the List

Next, change the pointer to the head of the list to point to the new node
Code for This

• new is a pointer to the new node, headList points to the head of the list
• First, make new point to the old head of the list
  new->next = headList;
• Next, make the pointer to the head of the list point to new
  headList = new;
Insertion: In the Middle of the List

First, scan down the list until you reach the node before which the new node goes.

headList

new node goes after this one
Insertion: In the Middle of the List

Change the pointer in the new node to point to the first node after where the new node is to go.

new node goes after this one
Insertion: In the Middle of the List

Next, have the pointer in the node *before* where the new node is to go point to the new node.

new node goes after this one
Code for This

• new is a pointer to the new node, headList points to the head of the list, and p is a pointer to node
• First, find the node that new goes after
  for (p = headList;
      p != NULL && p->next < new->next;
      p = p->next)
  /* do nothing ;
• Next, change the pointer in new to point to the node after where this one goes
  new->next = p->next;
• Finally, make the node p points to point to new
  p->next = new;
Insertion: At the End of the List

First, scan down the list until you reach the end node.

new node goes after this one
Insertion: At the End of the List

Next, change the pointer in the end node to point to the new node

new node goes after this one
Code for This

• new is a pointer to the new node, headList points to the head of the list, and p is a pointer to node
• First, find the node at the end
  for (p = headList;
       p != NULL && p->next != NULL;
       p = p->next)
    /* do nothing */;
• Next, change the pointer in what p points to to point to new
  p->next = new;
• This may be an excess, but make sure new’s pointer field is NULL
  new->next = NULL;
Multiple Arrays

• Need to store several data of different types about something
• Example: sort planets by their diameters
• Use 2 arrays
  • char *names[9]
  • int diameters[9]
• When sorting, need to keep both arrays aligned
  • So when swapping 2 elements of array diameter, the corresponding elements of array names must also be swapped
• Alternate approach: use structures!
Same with Structures

• Instead of 2 arrays, combine into one structure for each element, and use an array of structures

```c
struct celestial {
    char *name;  /* pointer to name of planet */
    int diameter;  /* diameter of planet in km */
} planets[9];
```

• This allocates space for 9 planets

• When you swap elements, you only need to swap one, not two, as in the parallel arrays case
And now a Word About argv

```c
void main(int argc, char *argv[]) {
    // Program name is argv[0]
    // One way to go down the arguments (j is declared as int j):
    for(j = 1; j < argc; i++)
        printf("Argument: \%s\n", argv[1]);
    // And the same thing, but using pointers (a is declared as char **a):
    for (a = argv+1; *a; a++)
        printf("Argument: \%s\n", *a);
}
```