C Programming - Lecture 4

- This lecture we will learn:
  - Why I harp on about pointers.
  - Arrays of pointers.
  - What are command line arguments.
  - The difference between an array and a pointer.
  - Why pointers are dangerous as well as confusing.
  - All the rest of the C language.

What's the point of pointers?

- Pointers can be used as variable length arrays.
- Pointers can be used for advanced data structures.
- Pointers can be "cheaper" to pass around a program.
- You could program without using them but you would be making life more difficult for yourself.
- Some things simply can't be done sensibly in C without them.

Arrays of pointers

- More commonly used (by experienced programmers) is an array of pointers.
- We can use an array of pointers in a similar way to a multi-dimensional array.
- We can declare an array of pointers like so:

```
char *name[]= {"Dave","Bert","Alf"};
/* Creates and initialises 3 names */
```

We can now use `name[0]` anywhere we could use a string.

Example of pointer arrays

```
int i;
char *people[]= {"Alf","Bert","Charlie"};
for (i= 0; i < 3; i++} {
    printf ("String %d is %s\n",i+1,people[i]);
}
```

Will print:

```
String 1 is Alf
String 2 is Bert
String 3 is Charlie
```

Example of pointer arrays (2)

```
int *lists[100];  /* Get 100 ptrs to int */
int i;
for (i= 0; i< 100; i++) {
   lists[i]= (int *)malloc(23*sizeof(int));
}
/* Do something with them here */
 for (i= 0; i < 100; i++) {
    free(lists[i]);
}
```

What are Command line arguments?

- In unix we can type, for example,

```
mv myfile1.dat myfile2.dat
```

- In windows we can do similar things though it is more tricky (and less commonly used).
- The myfile1.dat and myfile2.dat are known as command line arguments
- But how would we write a C program which could listen to command line arguments

Arguments to main introduce command line arguments

- Instead of writing

```
int main()
```

- Write

```
int main(int argc, char *argv[])
```

- argc tells us the number of command line arguments
- argv[0] is the first, argv[1] is the second, argv[2] is the third etc.

Example code to print arguments

```
int main (int argc, char *argv[])
{
    int i;
    for (i= 0; i < argc; i++) {
        printf ("Argument %d is %s",i+1,
            argv[i]);
    }
    return 0;
}
```

Using command line arguments

```
int main(int argc, char *argv[])
{
    FILE *fptr;
    if (argc < 2) {
        /* Do some error handling */
    }
    fptr= fopen (argv[1],"r");
    if (fptr == NULL) {
        /* Do some other error handling */
    }
    return 0;
}
```

What's the difference between an array and a pointer

- Arrays have memory already allocated.
- We cannot move an array to point at something else.
- There's no such thing as a multi-dimensional pointer (but you can fake one).

The true HORROR of pointers

- Pointer programming allows more sophisticated programming technique.
- But if we mess up with pointers, we really mess up.
- It is best to only use pointers once you are confident with simpler programming techniques.
- Pointer bugs are the hardest to find - you might find your program crashes randomly at different points in the code. This is typical of pointer bugs.

Writing to unassigned memory

```
int *a;
*a= 3;   /* Writes to a random bit of memory */

int *a;
a= malloc (100*sizeof(int));
/* malloc memory for 100 ints */
a[100]= 3;  /* Writes to memory off the end of the array */

int *a;
a= malloc (100*sizeof(int));
/* malloc memory for 100 ints*/
.
.  /* Do some stuff with a*/
.
free (a);                    /* free it again */
.
.  /* Do some other stuff during which we forget a is freed */
.
*a= 3;    /* Writes to memory which has been freed – very bad*/
```

Memory leaks

- If we allocate memory but don't free it this is a "memory leak"

```
void my_function (void)
{
    int *a;
    a= malloc(100*sizeof(int));
    /* Do something with a*/
    /* Oops – forgot to free a */
}
```

Every time we call this function it "steals" 100 ints worth of memory. As we call this function more the computers memory will fill up.

Rogue pointers

- Rogue pointers are pointers to unassigned memory. If we try to access rogue pointers we will be writing to unassigned memory

```
int *calc_array (void)
{
    int *a;
    int b[100];
    /* Calculate stuff to go in b */
    a= b;  /* Make a point to the start of b*/
    return a;   /* Ooops – this isn't good */
}
```

Pointers to pointers?

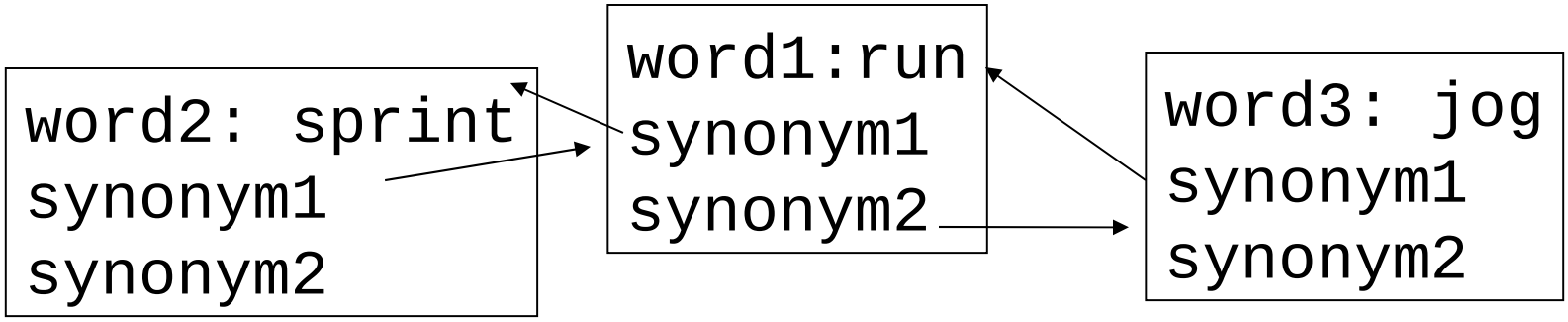
- We can also have pointers to pointers:

```
int number= 5;
int *ptrtonumber;
int **ptrtoptrtonumber;
ptrtonumber= &number;
ptrtoptrtonumber= &ptrtonumber;
*(*ptrtoptrtonumber)= 6;
```

- We can even have pointers to functions:
- If you want to use them then feel free.

Linked Lists

- Sometimes programmers want structs in C to contain themselves.
- For example, we might design an electronic dictionary which has a struct for each word and we might want to refer to synonyms which are also word structures.



How structs can contain themselves

- Clearly a struct cannot literally contain itself.

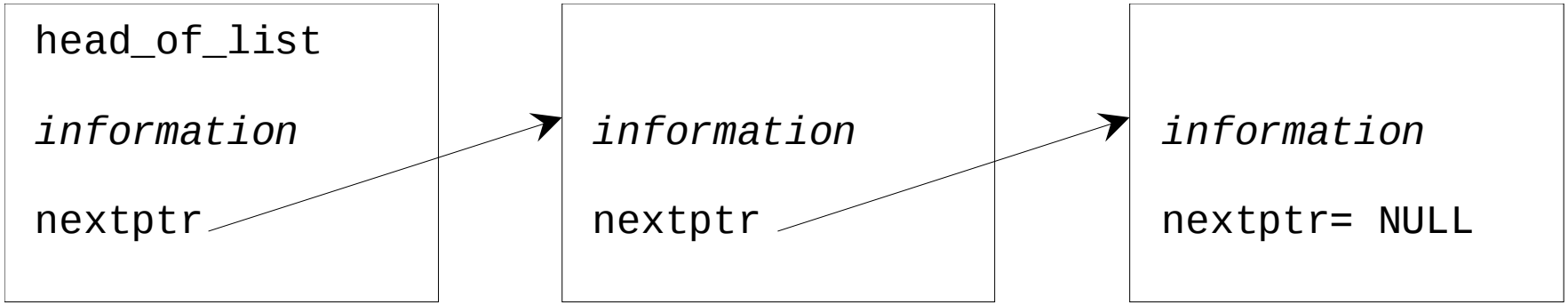
```
struct silly_struct {   /* This doesn't work */
    struct silly_struct s1;
};
```

- But it can contain a pointer to the same type of struct

```
struct good_struct { /* This does work */
    struct *good_struct s2;
};
```

The linked list - a common use of structs which contain themselves

- Imagine we are reading lines from a file but don't know how many lines will be read.
- We need a structure which can extend itself.

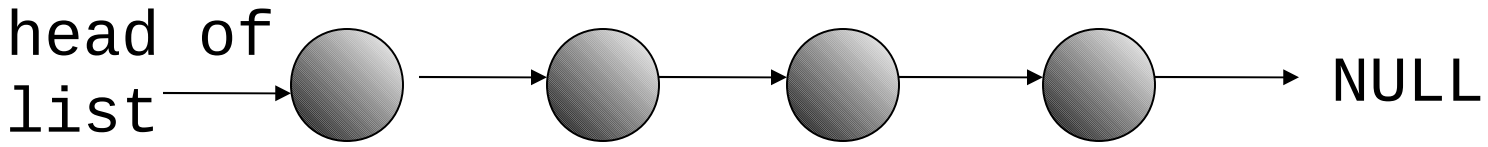


This is known as a linked list. By passing the value of the head of the list to a function we can pass ALL the information.

How to set up a linked list

```
typedef struct list_item {
    information in each item
    struct list_item *nextptr;
} LIST_ITEM;
```

This structure (where information is what you want each "node" of your linked list to contain). It is important that the nextptr of the last bit of the list contains NULL so that you know when to stop.

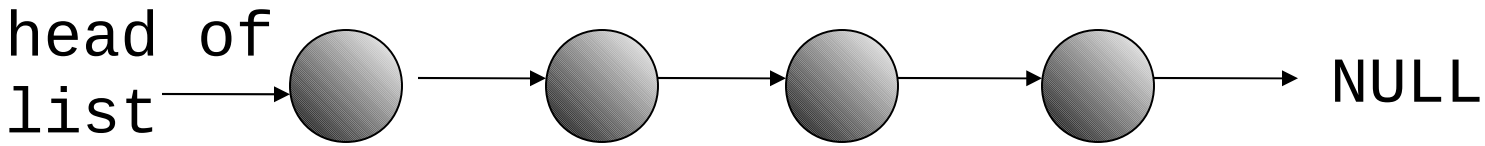


Address book with linked lists

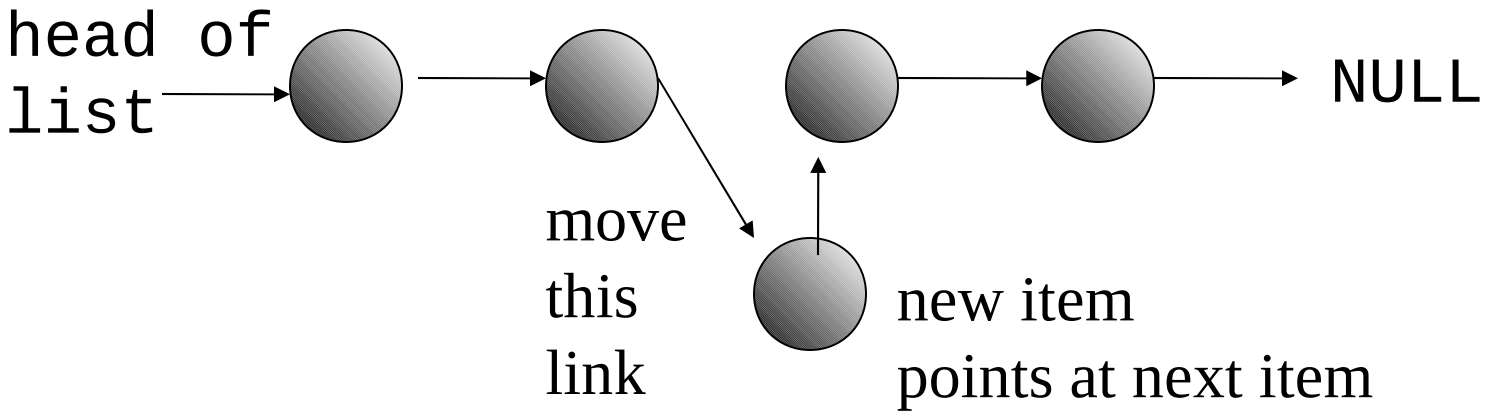
```
typedef struct list {
    char name[MAXLEN];
    char address[MAXLEN];
    char phone[MAXLEN];
    struct list *next;
} ADDRESS;


ADDRESS *hol= NULL;
/* Set the head of the list */
```

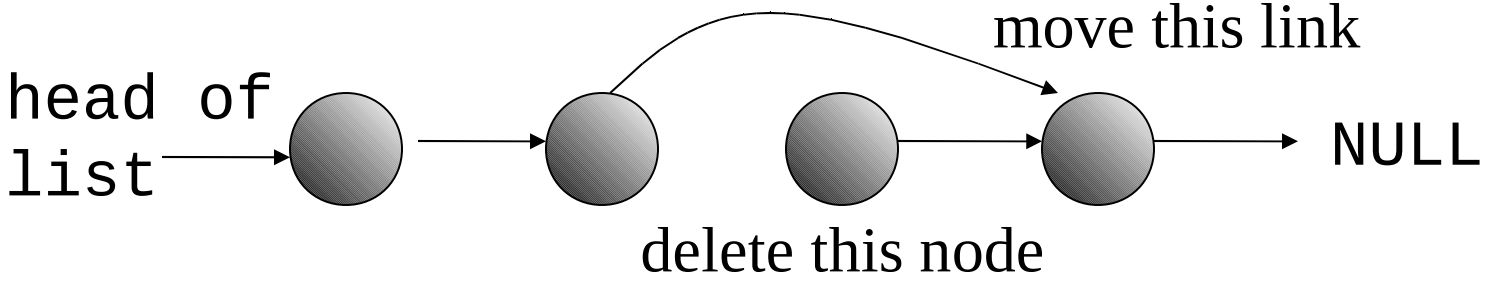
Linked list concepts



Adding an item to the middle of the list



Deleting an item from the middle of the list



Adding to our address book

```
void add_to_list (void)
/* Add a new name to our address book */
{
    ADDRESS *new_name;
    new_name= (ADDRESS *)malloc (sizeof (ADDRESS));
   /* CHECK THE MEMORY! */
    printf ("Name> ");
    fgets (new_name->name, MAXLEN, stdin);
    printf ("Address> ");
    fgets (new_name->address, MAXLEN, stdin);
    printf ("Tel> ");
    fgets (new_name->phone, MAXLEN, stdin);
    /* Chain the new item into the list */
    new_name->next= hol;
    hol= new_name;
}
```

Keywords of C revisited

- Flow control (6) – `if, else, return, switch, case, default`
- Loops (5) – `for, do, while, break, continue`
- Common *types* (5) – `int, float, double, char, void`
- *structures* (3) – `struct, typedef, union`
- Counting and sizing things (2) – `enum, sizeof`
- Rare but still useful *types* (7) – `extern, signed, unsigned, long, short, static, const`
- Evil keywords which we avoid (1) – `goto`
- Wierdies (3) – `auto, register, volatile`