

# *Lesson #8*

*Structures*  
*Linked Lists*  
*Command Line Arguments*

# *Introduction to Structures*

- ◆ Suppose we want to represent an atomic element. It contains multiple features that are of different types. So a single variable or even an array are inadequate to represent it. The solution is a structure that allows us to aggregate variables of different types under one logical name.

# *Creating a new type*

- ◆ By creating a structure, we create a new type of data. We know about `int`, `double` and `char`, let's create a new type called *struct element*.

```
struct element
{
    char name [10];
    char symbol [5];
    double atom_weight;
    int atom_number;
}
```

# *Creating a new type*

- ◆ The declaration of new types usually happens before the main program, just after the preprocessor directives so that the new types are known in all the functions of the program.
- ◆ Inside a function though, it is necessary, as for the other usual types to declare variables.

# *Declaring variables*

```
int a, b[3];  
struct element e1, e2, e3, e[50];  
are all valid declarations
```

- ◆ To fill the values into the components, we use the . (dot) operator. Components (dotted variables) behave exactly like regular variables.

```
strcpy (e1.name, "hydrogen");  
strcpy (e1.symbol, "H");  
e1.atom_number = 1;  
e1.atom_weight = 1.00794;
```

# The *typedef* Construct

- ◆ The typedef construct provides the possibility to define synonyms to built-in or user-defined data types.
- ◆ For instance, by having **typedef int age;** defined after the preprocessor directives, I create a new type *age* that is exactly like an integer. So I will be able to declare a variable **age x;** where x will be in reality an integer. I could also use **typedef struct element ele;** and for now on, we would be able to declare variables with *e/e* only: **ele e5;**

# *typedef + struct*

- ◆ Quite often we combine the structure definition with the user-defined data type alias associated with it.

```
typedef struct element
{
    char name [10];
    char symbol [5];
    double atom_weight;
    int atom_number;
}ele;
```

## *Placing values in structured variables*

1. We can declare and initialize and the same time:

```
ele e6 = {"hydrogen", "H", 1.00794, 1};
```

The order **must** reflect the structure definition.

2. We can fill the variables by assignment (see three slides back).

3. We can ask the user or read from a file.Ex:

```
scanf ("%lf", &e2.atom weight);  
fscanf (in, "%s", e2.symbol);  
gets (e2.name);  
fgets (e3.name, sizeof(e3.name), in);
```

# *Structures and functions*

- ◆ When a structured variable is passed as an input to a function, all of its component values are copied into the components of the function's corresponding formal parameters.
- ◆ Unlike arrays that require pointers to be passed to functions, structures can be passed by value. Of course you can use pointers also if you need multiple results, like we have seen before.

# *Structures and functions*

- ◆ Let's have a function that prints out a report on the element.

```
void  
print_element (ele e)  
{  
    printf ("Name: %s\n", e.name);  
    printf ("Symbol: %s\n", e.symbol);  
    printf ("Weight: %.11f\n", e.atom_weight);  
    printf ("Atomic Number: %d\n", e.atom_number);  
}
```

- ◆ In the main it would be called like  
`print_element (e1);`

# *Structures and functions*

- ◆ Let's have a function that compares two elements by comparing the two atomic numbers. It returns true if they are identical.

```
int
compare_elements (ele e1, ele e2)
{
    int equal;
    equal = e1.atom_number == e2.atom_number;
    return (equal);
}
```

- ◆ In the main it would be called like  
`compare_elements (e1, e[4]);`

# *Structures and functions*

- ◆ Let's have a function that fills a structure from the keyboard and then returns it back to the main.

```
ele
read_element (void)
{
    ele e;
    printf ("Enter the name, symbol, atomic
            weight and atomic number separated by
            spaces: ");
    scanf ("%s%s%lf%d", e.name, e.symbol,
            &e.atom_weight, &e.atom_number);
    return (e);
}
```

- ◆ In the main it would be called like  
`e1 = read_element( );`

# Structures, functions, and pointers

- ◆ Let's have a function that does exactly the same thing as the previous one except that it will read **two elements** and use a pointer parameter for the extra "fake result."

```
ele read_element (ele* eptr)
```

- ◆ Before going further we need to understand the order of pointer operations. If a pointer variable of type **ele** is called **eptr** then to fill its atomic number component, for example, we would be tempted to use `scanf ("%d", &*eptr.atom_number);` (notice that *eptr* is a pointer, the element itself is *\*eptr*). That would be a mistake, however, as we will see on the next slide.

# *Structures, functions, and pointers*

- ◆ The scanf statement on the previous slide is not correct because the dot (.) operator is always processed before the star (\*) operator. That would result in accessing the atomic number of a pointer (eptr.atom\_number) instead of the atomic number of an element.
- ◆ Therefore we need to use **scanf ("%d", &(\*eptr).atom\_number);** instead. There is a short cut in C for (\*eptr)., it is **eptr->**, therefore, we can use **scanf ("%d", &eptr->atom\_number);** as an alternative.

# Structures and functions

◆ Now let's see the function:

```
ele
read_element2 (ele* eptr)
{
    ele e;
    printf ("Enter the name, symbol, atomic
    weight and number separated by spaces: ");
    scanf ("%s%s%lf%d", e.name, e.symbol,
            &e.atom_weight, &e.atom_number);

    printf ("Enter the name, symbol, atomic
    weight and number separated by spaces: ");
    scanf ("%s%s%lf%d", eptr->name, eptr->symbol,
            &eptr->atom_weight, &eptr->atom_number);
    return (e);
}
```

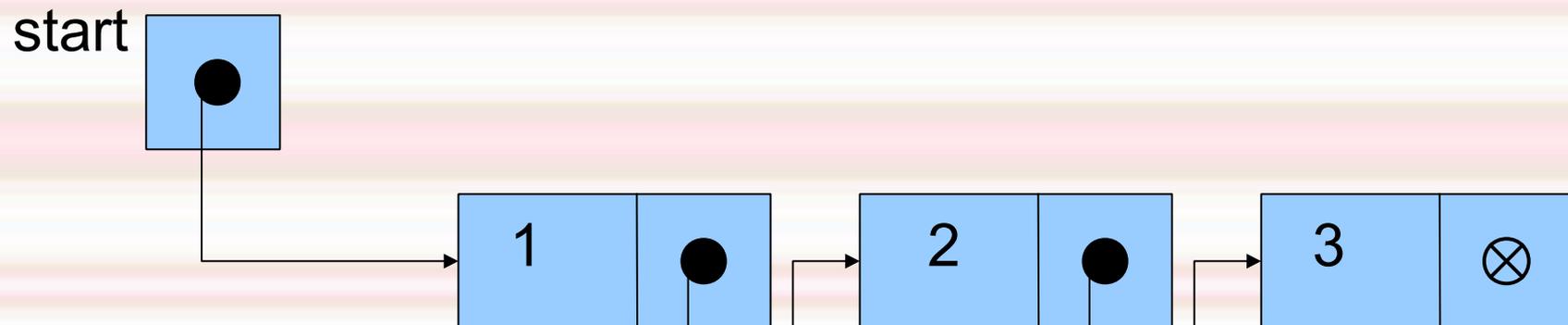
◆ In the main it would be called like  
`e1 = read_element2(&e2);`

# *Linked Lists*

- ◆ A list is a finite sequence, with order taken into account.
- ◆ A linked list consists of ordered nodes like  $N_1, N_2, N_3, \dots, N_n$  together with the address of the first node  $N_1$ .
- ◆ Each node has several fields, one of which is an *address field*.

# Linked Lists

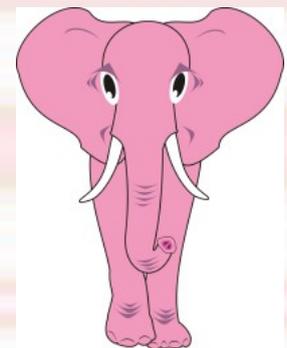
- ◆ The address field of N1 contains the address of N2, the address field of N2, the address of N3, and so on.... The address field of Nn is a *NULL* address. In the following example, the variable start contains the address of the first node. Each node has two fields, the first contains an integer indicating the position of the node in the list, the second field contains the address of the next node.



# *Example of a linked list*

- ◆ To declare a linked list, we need to declare a structure where the last component will be a pointer to a variable of the structured type itself.
- ◆ Let's define a list of elephants. Notice that the *next* component is a pointer to a *struct ELEPHANT*. The other component store the animal's name and weight.

```
typedef struct ELEPHANT
{
    char name [10];
    int weight;
    struct ELEPHANT* next;
}elephant;
```

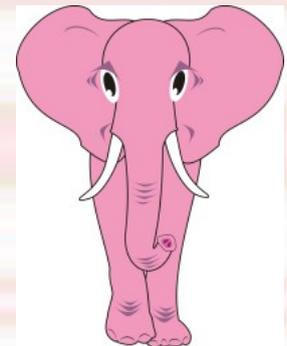


# *Example of a linked list*

- ◆ Next, let's declare variables to store our list nodes and the start of the list.

```
/* the nodes - ready to contain 3  
elephant names */  
elephant eleph1, eleph2, eleph3;
```

```
/* the start variable - pointer to an  
elephant node */  
elephant* start;
```

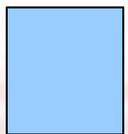


# Example of a linked list

- ◆ Let's fill now the elephants' names and weight:

```
strcpy (eleph1.name, "Edna");  
strcpy (eleph2.name, "Elmer");  
strcpy (eleph3.name, "Eloise");  
eleph1.weight = 4500;  
eleph2.weight = 6000;  
eleph3.weight = 4750;
```

start



eleph1

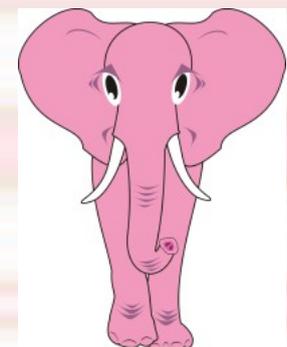
"Edna"	4500	
--------	------	--

eleph2

"Elmer"	6000	
---------	------	--

eleph3

"Eloise"	4500	
----------	------	--

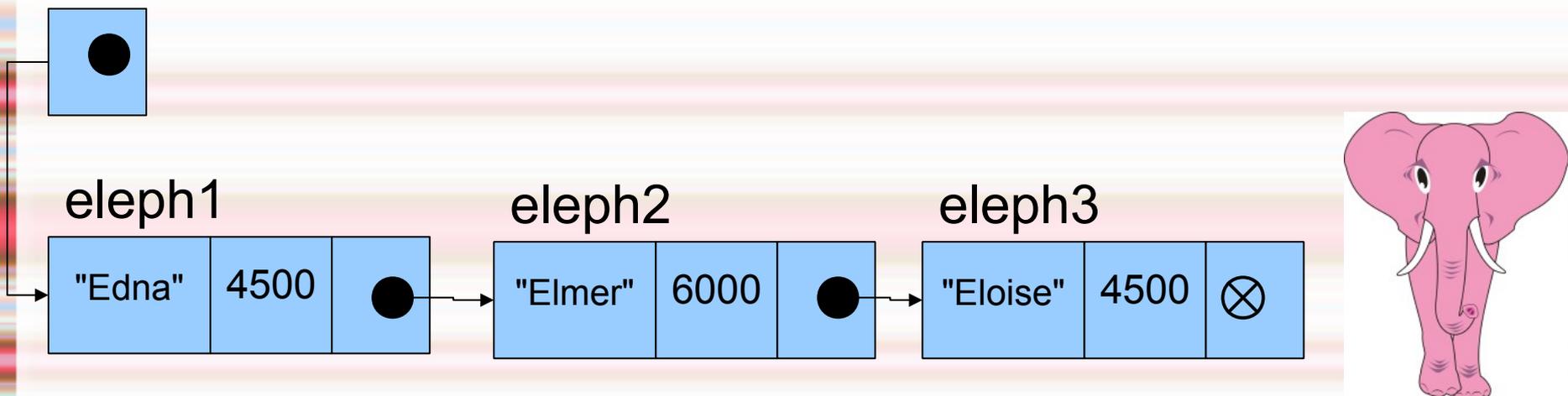


# Example of a linked list

- ◆ Next, let's link all these elephants together and build the linked list.

```
start = &eleph1;  
eleph1.next = &eleph2;  
eleph2.next = &eleph3;  
eleph3.next = NULL;
```

start

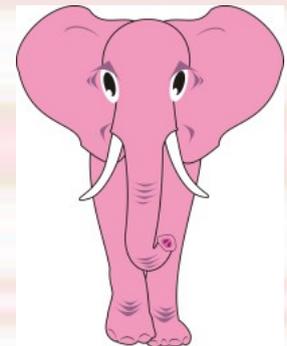


# *Example of a linked list*

- ◆ Finally, let's use our linked list. We will print out the names of all the elephants in it. We will use a variable  $p$  (`elephant* p;`) as a travelling variable through the list (not unlike the  $i$  variable we used to travel through an array).

```
count = 1; /* to count the elephants */

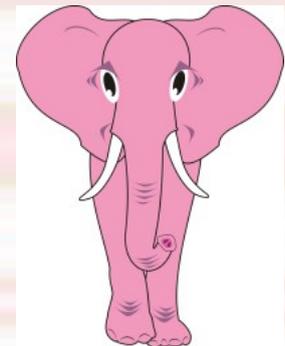
/* the list loop */
for (p = start; p != NULL; p = p->next)
{
    printf ("Elephant #%d is %s.\n",
count, p->name);
    count = count + 1;
}
```



# *Linked Lists and functions*

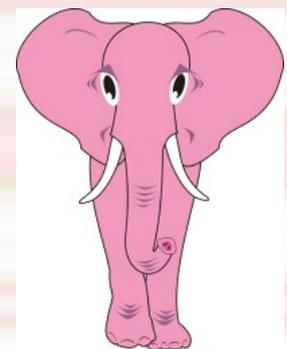
- ◆ Linked lists can of course be sent to functions. Let's have a function that accepts a list of elephants and returns 1 if an elephant weighing more than 5000kg can be found in the list.

```
int  
find5000 (elephant* ptr)  
{  
    int found = 0;  
    for (p = start; p != NULL; p = p->next)  
        if (ptr->weight > 5000)  
            found = 1;  
  
    return (found);  
}  
find5000 (start); would return 1 .
```



# *Lists and dynamic allocation*

- ◆ So far we knew in advance the number of elephants in the list. But what if we didn't? Let's now have a function named **get\_elephants** that will fill the list from user input. Since we do not know how many elephants the user will enter, we will use dynamic allocation for every node.
- ◆ In the main program, in that case, only the start variable will be declared and all the nodes will be dynamically allocated and filled with values in the function. The function will be called this way:
  - ◆ `start = get_elephants ( );`



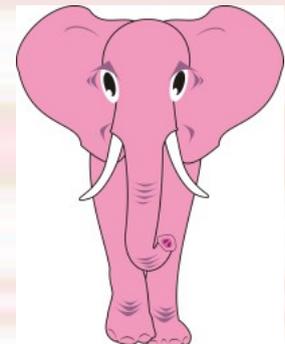
# *Lists and dynamic allocation*

```
elephant*
get_elephants (void)
{
    elephant *current, *first;
    int response;
    /* create first node */
    first = (elephant*)calloc(1, sizeof(elephant));
    current = first;

    printf("Elephant name? ");
    scanf ("%s", current->name);
    printf("Elephant weight? ");
    scanf ("%d", &current->weight);

    printf("\nAdd another? (y=1/n=0)");
    scanf ("%d", &response)
```

cont...



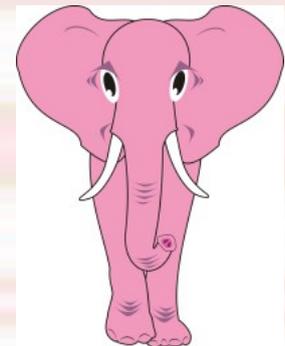
# *Lists and dynamic allocation*

```
while (response) /*while response is 1 (yes) */
{
    /* allocate node and change current pointer */
    current->next = (elephant *)calloc
(1,sizeof(elephant));
    current = current->next;

    /* fill node */
    printf("Elephant name? ");
    scanf ("%s", current->name);
    printf("Elephant weight? ");
    scanf ("%d", &current->weight);

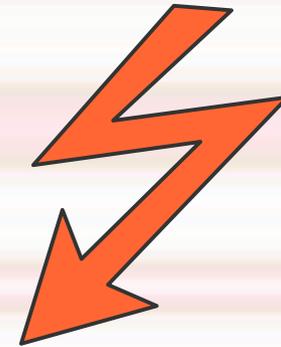
    printf("\nAdd another? (y=1/n=0)");
    scanf ("%d", &response);
}
```

cont...



# Lists and dynamic allocation

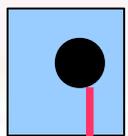
```
current->next = NULL;  
return (first);  
}
```



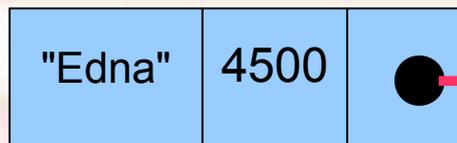
→ Now can you think how to:

- Add a node at the end of a linked list?
- Add a node at the beginning?
- Insert a node between two nodes?
- Delete a node?

start



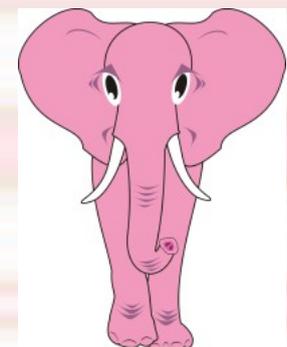
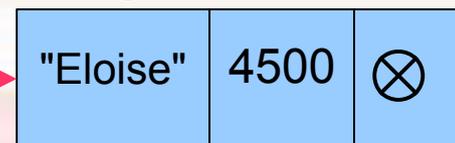
eleph1



eleph2



eleph3



See [ihypress.net/programming/c/10.php](http://ihypress.net/programming/c/10.php) for more examples.

# *int main (void)?*

- ◆ All proper C main programs begin with *int main (void)*. A main program is actually a function that takes arguments from the operating system and returns an integer (0).
- ◆ So far, we did not use any arguments in the main program because we did not send anything to the program from the operating system. That is why we used **void** as the sole argument for the main function.
- ◆ If we want to send information from the operating system to the program, we need to have command-line arguments.

# Command-line arguments

The main program can have two command-line arguments:

1. **argc**: an integer, representing the number of arguments (always at least 1).
  2. **argv**: an array of strings containing the arguments themselves. All arguments coming from the operating system are strings.
- ◆ So instead of **int main (void)**, the program header becomes  
**int main (int argc, char\* argv[ ])**

# Using command-line arguments

```
#include <stdio.h>
int
main (int argc, char* argv[])
{

    int i;

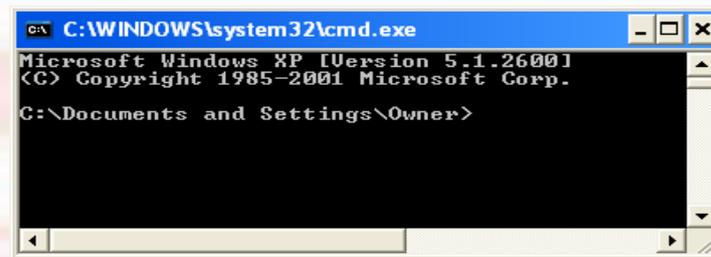
    /* argument 0 identifies the program */
    printf ("%s\n", argv[0]);

    /* the other arguments */
    for (i = 1; i < argc; ++i)
        printf ("%s ", argv[i]);

    return (0);
}
```

# *The Command Line*

- ◆ Now that we have seen how command-line arguments are processed by C programs, let's see now how to send the command lines.
- ◆ Command lines are not written in C, they are direct text commands to the operating system.
- ◆ To access the OS commands in Windows XP or Vista, we use Start > Run > **cmd**



# The Command Line

- ◆ Better still, we can use the command mode directly from Quincy.

Tools > Options > Run > Prompt each time...

- ◆ Then, when we run the program, we will be prompted for the command line.
- ◆ Let's run the program from two slides back.

Command line:

This is a command line

argv[0]

argv[1] to argv[5]

C:\ Quincy 2005

F:\c1a1.exe

This is a command line

Press Enter to return to Quincy...

# *Numerical arguments / atoi()*

- ◆ What if we needed to send numerical values into the command line?
- ◆ Since command line arguments are strings, we will need to convert those strings to numbers. For that we will need two new functions (need to include `stdlib.h`).
- ◆ The `atoi()` function converts a string into an integer, and returns that integer. The string must of course some sort of number, and `atoi()` will stop reading from the string as soon as a non-numerical character has been read (+ and – are considered numerical at the beginning of the string).
- ◆ `atoi("45.78");` will return the integer 45.

# *Numerical arguments / atof()*

The function `atof()` converts a string into a double, then returns that value. The string must start with a valid number, but can be terminated with any non-numerical character, other than **E** or **e** (+ and – are considered numerical at the beginning of the string).

- ◆ `atof( "42.57" );` will return the double 42.57.
- ◆ `atof( "45.3e3" );` will return the double 45300.0.
- ◆ See [ihypres.net/programming/c/12.php](http://ihypres.net/programming/c/12.php) for more examples.

# *The End*

