KU LEUVEN

C: an introduction

Structures: more

Pointers to Structures

- Defining pointers is the same as for variables of primitive types

```
struct Personnel {
        char name[100];
        int age;
        double height;
         }
  struct Personnel captain = {"Fred", 37, 1.83};
 struct Personnel * pp;
  pp = &captain;
 (*pp).age = 38; /* captain.age is now 38. */
```

- Notice the parentheses around the dereferenced pointer.
  - This is necessary to enforce correct precedence

Pointer to Member Operator ->

- An alternative notation permits simpler pointer access to structure members.

```
pp->age = 38; /* equivalent access operation */
```

- Another example,

```
struct Payroll lieutenant, *ppr = &lieutenant;

lieutenant.person.age = 23; /* equivalent operations */
(*ppr).person.age = 23;
ppr->person.age = 23;
```


Arrays of Structures

- The definition of arrays of structure types is the same as for arrays of primitive types.

```
struct Personnel pa[10];
```

- Structure arrays can be initialised with an initialiser list enclosed in braces
  - the size is determined by the compiler if unspecified

```
struct Personnel pa[] = {
  {"Fred", 37, 1.83}, {"Mary", 21, 1.65},
  {"Joe", 19, 2.1},   {"Cyril", 28, 1.71}
 };
```

Structures and Pointer Arithmetic

- As with primitive types, the compiler knows the size of a structure
  - The size of a structure in bytes may be found via the **`sizeof`** operator.
  - Pointer arithmetic on a structure pointer will compute the appropriate address offsets automatically
- Arrays of structures behave like arrays of primitive types

```
struct Personnel pa[] = {
          {"Fred", 37, 1.83}, {"Mary", 21, 1.65},
          {"Joe", 19, 2.1},   {"Cyril", 28, 1.71}
  };

struct Personnel * pp;
int i;

for (pp = pa; pp != pa + sizeof(pa)/sizeof(pa[0]); ++pp)
          printf("%s %d\n", pp->name, pp->age);
```


Self-Referential Structures

- A structure may not contain an object of its own type.

```
struct Node {
    int item;
    struct Node n; /* Invalid */
};
```

- In general, a structure may not contain an object of an *incomplete type*.
- However, a structure may contain a *pointer* to an incomplete type.

```
struct Node {
    int item;
    struct Node * pn; /* Valid */
};
```

Self-Referential Structures

- The ability to refer to (ie, point to) an incomplete type, including itself, is an important property for constructing a variety of data-structures.
- For example: linked-lists, binary trees, graphs, hash tables, and more.

Example: A Linked List

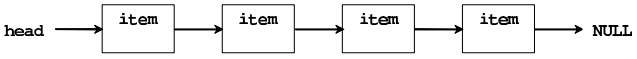
- Linked lists come in two basic varieties: singly linked and doubly linked.
- We describe here a simple version of a singly linked list.
- List consists of a set of nodes, where each node contains an item and a pointer to another list node.

```
struct List {
    int item;
    struct List * next;
};
```

- (Here we have chosen an `int` as the contained item. Any other type(s) may be used.)
- *File: demo_llist_1.c*
- *File: demo_llist_2.c*

Singly Linked List

- List is formed by connecting the pointer of one node to the address of the next.
- We keep a pointer to the *head* of the list. This permits traversal.
- The end of the list is marked by a NULL pointer.



- Example, to start at the head of the list and traverse to the end node:

```
struct List *node = head;
while (node->next != NULL)
    node = node->next;
 printf("Last node item: %d", node->item);
```



```c
#include <stdio.h>
// demo_llist_1.c
struct list
{
  int         numb;
  struct list *next;
};

void main()
{
                struct list n1, n2, n3, *pll;
                int         i, j;

                pll = &n1;
   printf(" pointer to n1 %p \n", pll);
                n1.numb = 100;
                n2.numb = 200;
                n3.numb = 300;
                n1.next = &n2;
                n2.next = &n3;
                i = n1.next->numb;
   j = n1.numb;

                printf("n2.next->numb= %d \n", n2.next->numb);
                printf("i(n1.next->numb)= %d \n", i);
                printf("j (n1.numb)= %d \n", j);
                printf("*pll->numb= %d \n", pll->numb);
}
```

```
frankvp@CRD-L-08004:.../Structures$ gcc -demo_llist_1.c
gcc: fatal error: no input files
compilation terminated.
frankvp@CRD-L-08004:.../Structures$ gcc demo_llist_1.c -o demo_llist_1
frankvp@CRD-L-08004:.../Structures$ ./demo_llist_1
 pointer to n1 0x7ffff5219470
n2.next->numb= 300
i(n1.next->numb)= 200
j (n1.numb)= 100
*pll->numb= 100
frankvp@CRD-L-08004:.../Structures$
```

```c
/*
demo_llist_2.c
build dynamically a list and fill it up with numbers
*/

#include<stdlib.h>
#include<stdio.h>

struct list_el {
   int val;
   struct list_el * next;
};

typedef struct list_el item;

void main() {
   item * curr, * head;
   int i;

   head = NULL;

   for(i=1;i<=10;i++) {
      curr = (item *)malloc(sizeof(item));
      curr->val = i;
      curr->next  = head;
      head = curr;
   }

   curr = head;

   while(curr) {
      printf("%d\n", curr->val);
      curr = curr->next ;
   }
}
```

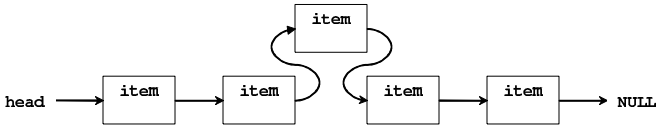
```
frankvp@CRD-L-08004:.../Structures$ gcc demo_llist_2.c -o demo_llist_2
frankvp@CRD-L-08004:.../Structures$ ./demo_llist_2
10
9
8
7
6
5
4
3
2
1
frankvp@CRD-L-08004:.../Structures$
```


Linked-List Properties

- Linked-Lists are useful because they can be grown (or shrunk) very easily. Unlike arrays, there are no issues of reallocating memory and copying data.
- Nodes can even be inserted (or removed) from midway along the list without difficulty (and efficiently).

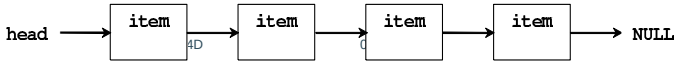
Adding Nodes to a List

- Adding a node to the end of the list.
- Adding a node midway through the list.



Splicing in a New Node

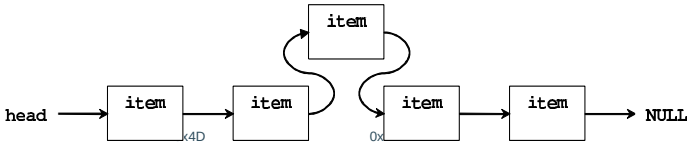
- Remember that everything is manipulated as an address. Consider the pointer variables, eg.,
  - `node` is address 0x4D
  - `node->next` is address 0xA1
  - `newnode` is address 0xB6



- Splicing:

```
newnode->next = node->next;     assign to 0xA1
node->next = newnode;           assign to 0xB6
```

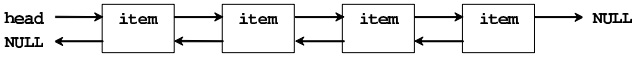
0xB6



Doubly Linked Lists

- A doubly linked-list is similar to a singly linked list except that each node additionally contains a pointer to the previous node.

```
struct List {
    int item;
    struct List *next;
    struct List *prev;
};
```



- Doubly linked-lists permit traversal both up and down the list. They tend to permit simpler and more efficient node deletion.
- *File: double_llist.c*