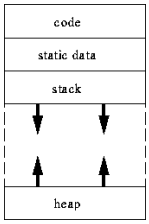
CS 201
Dynamic Data Structures

Debzani Deb

Run time memory layout

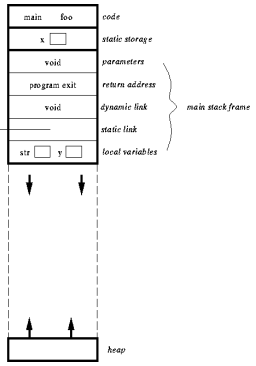
- When a program is loaded into memory, it is organized into four areas of memory
  - Code: The compiled code of the program, including all functions both user-defined and system.
  - Static data: global variables
    - Allocated during run time.
  - Stack:
    - Contains stack frame that includes local variables, parameters, return address etc.
    - Function call - push stack frame
    - Function exit - pop stack frame
  - Heap:
    - Dynamic memory – list of free space
    - Allocated by calloc(), malloc().
    - Deallocated by free().



Run time memory layout (example)

```
int x;                          /* static storage */
void main() {
    int y;                      /* stack storage */
    char *str;                  /* stack storage */
    str = malloc(100);          /* allocates 100
          bytes of dynamic heap storage */
    y = foo(23);
    free(str);    /* deallocates 100 bytes of
                     dynamic heap storage */
} /* y and str deallocated as stack frame is
     popped */

int foo(int z) {        /* z is stack storage */
    char ch[100];   /* ch is stack storage */
    if (z == 23)  foo(7);
    return 3; /* z and ch are deallocated as
    stack frame is popped, 3 put on top of
    stack */
}
```



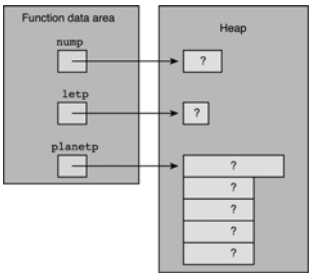
Dynamic memory allocation (1)

- It is not always known in advance how big an array will be in running program. There are two ways to go around this.
  - Set aside huge amount of memory when writing the program and hope that it is more than needed.
  - Allocate the needed memory "on the fly" from the heap.
- The later is obviously more intelligent and flexible.
- In order to allocate memory for any built-in or user-defined types, we do the following

```
nump = (int *) malloc (sizeof (int));
planetp = (planet_t *) malloc( sizeof (planet_t));
```

  - malloc returns a pointer that points to the start of the allocated memory (or NULL if there is not enough space in the heap).
  - The pointer returned by malloc has no type (pointer to void) and must be cast to the proper type.

Figure 14.4 Dynamic Allocation of Variables for an int, a char, and a Five-Component planet_t Structure



Dynamic memory allocation (2)

- If you need memory to store n array element, each of which is (say) an integer, then you need to do this

  a = (int *) malloc(total_array_size);

  where total_array_size = n * sizeof (int)
  - Alternative calloc (contiguous allocation function)

  a = (int *) calloc(n, sizeof(int));
  - calloc initializes the array element to zero, on the other hand malloc provides no initialization.
  - Both calloc and malloc are declared in <stdlib.h>.

Dynamic memory deallocation

- Memory allocated by calloc and malloc must be given back explicitly.
  - free(a) return to the heap the cell whose address is in a.
  - If a is NULL, there is no effect, if it is the base address of the space allocated by calloc and malloc, the space is deallocated, otherwise there is a system-dependent error.
  - free(planetp) returns the entire structure pointed to by planetp.
  - If you do not give the unused memory back, but keep on taking more, you will eat it all and at some point the program will crash.

Example

Types must match

```
a = (int *) malloc( n * sizeof(int)); /* allocate space for n-
                                         element array */
if (!a) {
   printf ("Memory allocation failed !\n");
   exit(1);
}
fill_array (a, n);          /* fill the array with numbers */
print_array (a, n);         /* print out the array */
printf ("Sum = %d\n", sum_array (a, n));
free(a);
```

Always make sure pointer is not Null

Always free allocated memory before exiting the program.

Pointer to void

- Pointer must always have a type that they point to.
- Exception: void * is a generic pointer, and can be assigned to any variable type.
- However, for good practice, you should always "type cast" it into the appropriate pointer type.
  - Example:

    int * p;

    void * v;

    If v happens to point to some integer, this is the proper assignment:

    p = (int *) v;

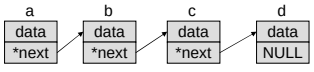
    This makes sure that the pointer types agree.

Linked list (1)

- A linked list is a sequence of nodes in which each node but the last contains the address of the next node.
- An alternative to arrays.
- Common abstraction to use in programming.

```
struct list {
    int data;
    struct list *next;
} a, b, c, d;
```

```
a.next = &b;
b.next = &c;
a.next -> next -> next  = &d;
/* an alternative c.next = &d; */
d.Next = NULL;
```



Linked list (2)

- To construct a dynamic linked list, we need to use nodes that have pointer components.
- Because we may not know how many elements will be in our lists, we can allocate storage for each node as needed and use its pointer component to connect it to the next node.
- Definition of nodes we are going to use in this lecture

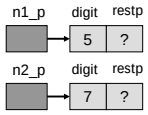
```
typedef struct list_node_s {
    int digit;
    struct list_node_s *restp;
} list_node_t;
```

  - In this definition, struct list_node_s is an alternative name for type list_node_t.

Linked list (2)

- We can allocate and initialize data components of nodes as follows

```
list_node_t *n1_p, *n2_p;
n1_p = (list_node_t *) malloc (sizeof(list_node_t));
n1_p -> digit = 5;
n2_p = (list_node_t *) malloc (sizeof(list_node_t));
n2_p -> digit = 7;
```



- Connecting nodes
  - Pointer components of above nodes are still undefined.

    n1_p -> restp = n2_p;

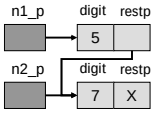
    copies the address stored in n2_p into the restp component of the node accessed through n1_p. Now they are connected.
- Accessing digit (7): two ways

```
n2_p -> digit;
n1_p -> restp -> digit;
```

- End the list

```
n2_p -> restp = NULL;
```

- n1_p is the pointer to the first element of the list and is called **list head**. If we know the address in n1_p, we can access every element of the list.



Link List Operation (Traversing)

- **Traversing**: Processing each node in a linked list in sequence, starting at the list head.
- Once you understand the following code, try to write a small function that counts the number of elements in a list.

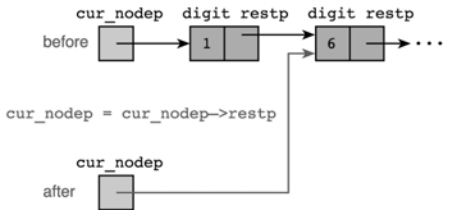
```
1.  /*
2.   *  Displays the list pointed to by headp
3.   */
4.  void
5.  print_list(list_node_t *headp)
6.  {
7.       if (headp == NULL) {    /* simple case - an empty list                  */
8.            printf("\n");
9.       } else {                /* recursive step - handles first element       */
10.           printf("%d", headp->digit);  /*      leaves rest to                 */
11.           print_list(headp->restp);    /*      recursion                      */
12.      }
13. }
```


Link List Operation (Traversing)

Comparison of recursive and iterative list printing

```
                    /* Displays the list pointed to by headp */
                    void
                    print_list(list_node_t *headp)
{                                       { list_node_t *cur_nodep;
 if (headp == NULL) {/* simple case */
     printf("\n");                        for  (cur_nodep = headp; /* start at
 } else {         /* recursive step */                              beginning  */
     printf("%d", headp->digit);                cur_nodep != NULL; /* not at
     print_list(headp->restp);                                      end yet    */
 }                                              cur_nodep = cur_nodep->restp)
}                                             printf("%d", cur_nodep->digit);
                                             printf("\n");
                                            }
```


Figure 14.19 Update of Iterative List-Traversing Loop Control Variable



Link List Operation (Searching)

Find First occurrence of target in the list.

```
1.  /*
2.   *  Searches a list for a specified target value.  Returns a pointer to
3.   *  the first node containing target if found.  Otherwise returns NULL.
4.   */
5.  list_node_t *
6.  search(list_node_t *headp,  /* input - pointer to head of list  */
7.         int          target)  /* input - value to search for      */
8.  {
9.      list_node_t *cur_nodep;  /* pointer to node currently being checked */
10.
11.     for  (cur_nodep = headp;
12.           cur_nodep != NULL  &&  cur_nodep->digit != target;
13.           cur_nodep = cur_nodep->restp) {}
14.
15.     return (cur_nodep);
16. }
```

1. What if I put cur_nodep++ instead of cur_nodep -> restp? Could that work? When?
2. What if the order of the following tests are reversed?
   (cur_nodep != NULL) && (cur_nodep -> digit != target)

Link List Operation (Insertion at Head)

- Insert at list's head (i.e. at the front of the list)

```
list_node_t * insertH (list_node_t * headp, int v) {
        list_node_t *newp;
        newp = (list_node_t *) malloc(sizeof(list_node_t));
        newp->digit = v;
        newp->restp = headp;
        return newp;  /* return pointer to the new head of the list */
}
```

```
int main(void){
        list_t * pHead = NULL;
        pHead = insertH(pHead, 3);
        pHead = insertH(pHead, 5);
        pHead = insertH(pHead, 7);
        pHead = insertH(pHead, 9);
}
```

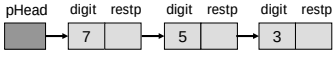
```
typedef struct list_node_s {
   int digit;
   struct list_node_s *restp;
} list_node_t;
```


Link List Operation (Deletion at Head)

- After the insertion at last slide, your list now looks like

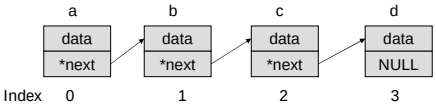

```
list_node_t * deleteH (list_node_t * headp){
        list_node_t *newp = headp -> restp;
        /* newp is now pointing to the 2nd element of the list */
        free(headp);
        return newp;
        /* return pointer to the new head of the list */
}
```

After the following call
pHead = deleteH (pHead);



Lists are arrays

- Quite often the lists are treated as arrays, that can change their size dynamically.



- Assumptions
  - Indexing of list starts at 0 (as in arrays).
  - Every index value is unique.
  - Indices are in growing order (incremented by 1).

Link List Operation (Deletion at Index)

- Delete element at some index of the list

```
list_node_t * deleteIndex (list_node_t * headp, int index) {
   int i;
   list_node_t * newp, *p, * tmpp;
   if (index == 0) {          // deleting head
        newp = headp -> restp;
        free(headp);
        return newp;
   }
```

Link List Operation (Deletion at Index)

```
else {                                  // deleting element other than the head
        for (p = headp, i = 1; (i < index) && (p -> restp != NULL); i++)
        /* searching for the element that has a pointer to the one to be
           deleted */
                p = p -> restp;
        tmpp = p -> restp;
        /* tmpp now points at the element to be deleted */
        if(tmpp != NULL)
                p -> restp = tmpp -> restp;
        /* p now points at the element after the one to be deleted */
        else
        /* we are dealing with the last element of the list */
                p -> restp = NULL;
        free(tmpp);
        return headp;
    }
}
```


Important !

- Final date has been posted in the website.
- Remember Exam 2 next Wednesday.
- Covers
  - Array
  - String
  - Recursion
  - Structure & Union
  - Dynamic data storage (upto next Monday)
- All rules of exam 1 will be applicable.