INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION

INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/643 - 18

SECOND COLLEGE ON
MICROPROCESSOR-BASED REAL-TIME CONTROL -
PRINCIPLES AND APPLICATIONS IN PHYSICS
5 - 30 October 1992

ADVANCED C
(Part I)

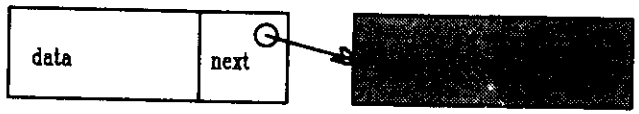
A. NOBILE
International Centre for Theoretical Physics
Trieste
Italy

SELF-REFERENTIAL STRUCTURES

LINKED LISTS

Structures cannot contain themselves as components
Structures can contain pointers to anything as components, even pointers to themselves

```
struct list_node{
    char[100] name;
    struct list_node *next;
}
```



- Contains *a pointer* to itself (allowed)
- **list_node** known as a *structure tag* as soon as encountered in first line, therefore **struct list_node *** is understood;

- example of *partial* or *incomplete declaration*: declare a *tag* to refer to a structure, then refer to it through pointers; complete declaration of structure before declaring any variable; can be used in general;

Example:

```
struct s1 ; /*incomplete */
struct s2 {
    int something;
    struct s1 * cross;
}
struct s2 {
    float something_else;
    struct s2 *cross2;
}
```

- **typedef** WOULD NOT WORK (ONLY place where *tags* NEEDED)

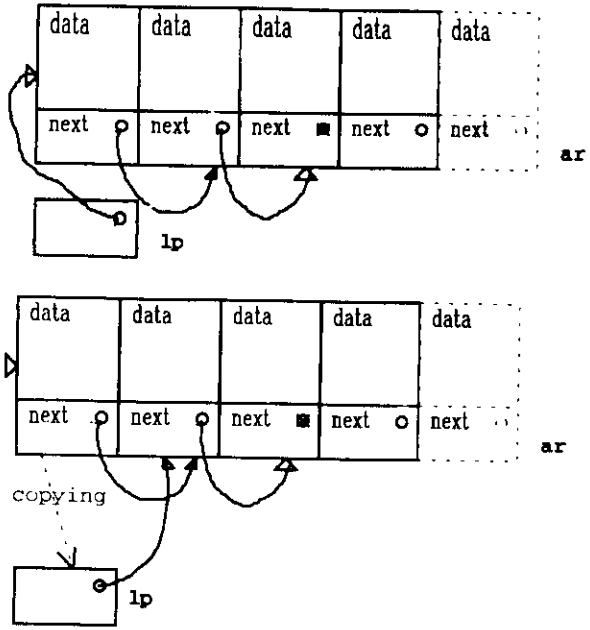
```
typedef { int data;
          ListElem *next;/*wrong:
            ListElem unknown here*/
} List_elem;/* type List_elem known
only after this point */
```

WARNING:
partial declaration is obtained just by mentioning name:

```
struct abc{ struct xyz *p};
/*struct xyz now partially declared
*/
```

DANGEROUS

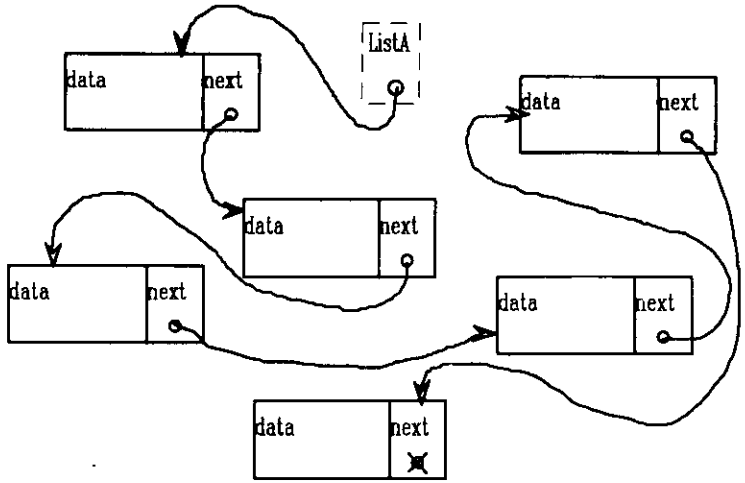

```
/* this program creates a linked
list and prints it out following
the pointers
 */

#include <stdio.h>
struct list_elem{
    int data;
    list_elem *next;
} ar [10];
main()
{
    struct list_ele *lp;

    ar[0].data = 5;
    ar[0].next = &ar[1];
    ar[1].data = 99;
    ar[1].next = &ar[2];
    ar[2].data = -7;
    ar[2].next = 0; /* null pointer
to next: end of list */

    lp = ar;
    while (lp! = NULL) {
        printf (" contents %d\n",
                lp->data)
                /* (*lp).data*/;
        lp = lp->next/*(*lp).next*/;
    }
    exit(0);
}
```



- move from one element to the next following pointers (lp=lp->next, NOT lp++)
- array structure not used at all

DYNAMIC OBJECTS

Lists are typical example: array structure not used.



Please note: most often, list referred to through a variable pointer to their first element:

```
list_elem *ListA;
```

Sometimes, through couple of variable pointers to first and last element (

```
struct list_id {
    struct list_elem *first, *last;
}ListA;
```

Both approaches help dealing with empty list case.

New problems:
add node to the list (for instance, to the end):

create a node
put new data in its **data** component
put a pointer to the new node in the (null) **next** component of the last element
put a **NULL** pointer in the **next** component of new node

delete a node from the end of a list:

detach the node from the list, putting a NULL in the **next** component of previous node

delete the node

CREATING an object of type ***T***

- obtain from the system enough memory to contain a copy of an object of type ***T***;
- handle that memory as if it was an object of type ***T***

```
struct list_ele *p;
int l_sz;

l_sz = sizeof( struct list_ele );
p=(struct list_ele *)malloc (l_sz);
```

malloc (size) **requires to the system to provide** a block of **size** bytes, and returns a pointer to this block (NULL if memory not available)
(struct list_ele *) is a *cast* that transforms the pointer returned by **malloc** int a pointer to **list_ele.**

```
p->data = new_value;
p->next = NULL;
last->next = p; /* assuming last
                   points to the last
                   element*/
```


COMMENT:
ANSI says **malloc** is of type **void *** meaning a pointer that can be casted to point to any type;
Old C says malloc is **char ***, but it returns a value suitable be casted to point to anything...;
In both cases casting does not cause any change in the returned pointer.

p useless:

```
if((last->next=(struct list_ele *)
       malloc(l_sz)) != NULL)
{
    last->next->next = NULL;
    last->next->data = new_value;
    last = last->next; /* keep it
       pointing to the last */
}else{.....
```

Deleting an object:

- Only if the object was created by `malloc`;
- `free(p)` /* p pointer to the object to be deleted */

Deleting the first element of a list

```
struct list_elem *ListA;

if (ListA) /* if list empty, do
              nothing */{
   struct list_elem *temp;
   temp= ListA;/*keep address of
              first node*/
   ListA=ListA->next; /*first node
              unlinked from ListA*/
   free(temp);/*delete old first
              node*/
}
```

Note 1: sequence is: unlink-free.
Note 2: Would the above be good as the body of a function that performs list-element removal?

WARNING:

```
struct list_ele *listA, *listB;
   listB = listA;
.............
   if (listA) /* if list empty, do
nothing */
   {  struct list_ele *temp;
      temp = listA;
      listA = listA->next; /*first
node unlinked*/
      free(temp);
   }
............
   listB -> data=.../* AAAARGGHHHH
*/
```

DANGLING POINTERS problem.

WARNING 2: Passing lists to functions:
Lists <=> pointers to their first elements
passing by reference if lists have to be be modified => passing pointer to lists => passing pointers to pointers to first element

```
function remove_first(List)
list_elem **List;
{ list elem *temp;
   if(*List){
   temp=*List;
   *List=(*List)->next;
   free (temp);
}
```

WARNING 3

```
while ( something){
    allocate memory
    use it
    forget it (without freeing)
}
```

causes problems difficult to trace (memory can get exhausted depending from path in the program ,data, etc.)

MORE EXAMPLES ON LISTS: A small list library

Header file:

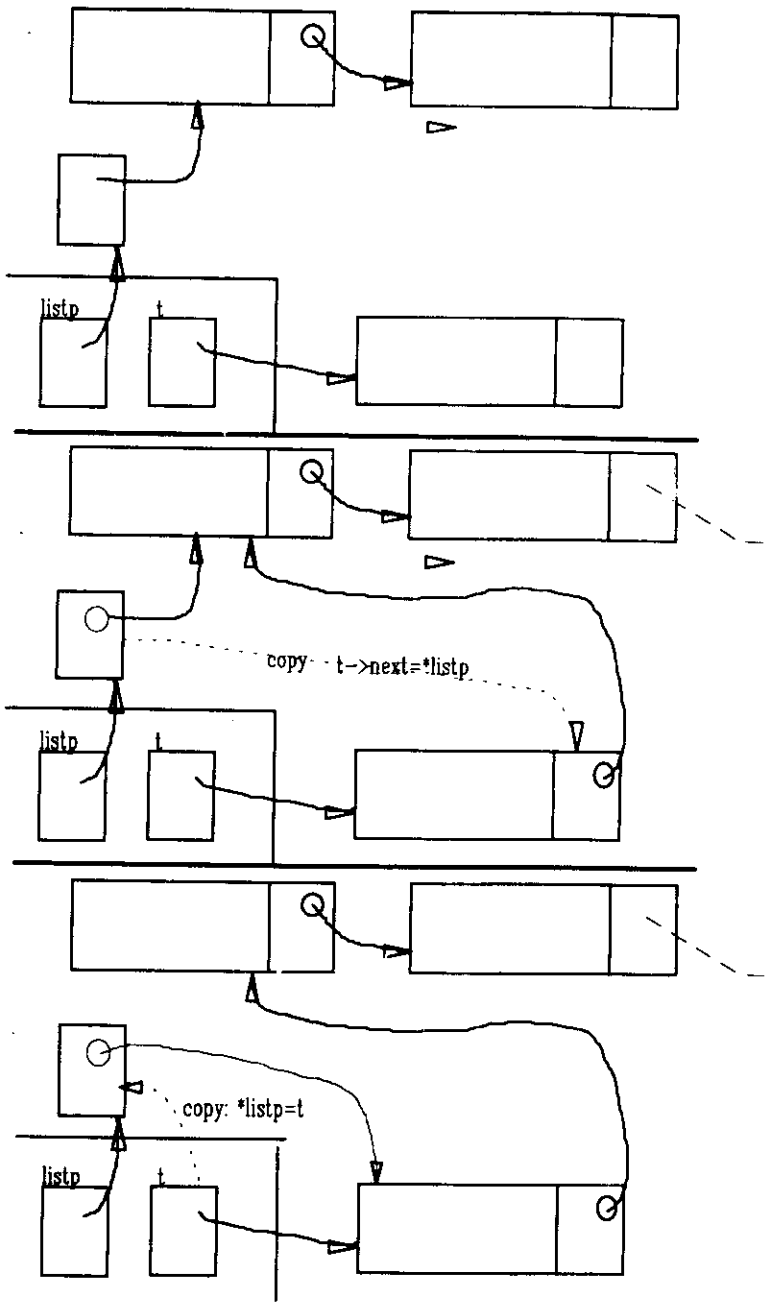
list.h

```
#ifndef _LISTH_
#define _LISTH_
/* before including this file,
typedef ListData,
typedef ListDataPattern;
when using the routines defined
here define function
ListDataCpy(a,b)
as a function copying an object
of type ListData from *b into *a
*/

struct ListElem { ListData data;
        struct ListElem *next;
};
typedef struct ListElem ListNode;
typedef ListNode * List;
List AddBfr();
/* ANSI of the above:
List AddBfr(ListData * d, List * lp);
adds a node containing data *d
before the list pointed by *lp */
List RmFrst();
/* List RmFrst(List *lp); removes
first element of lp*/
List AddSrt();
List RAddSrt();
/* List AddSrt(ListData *d, List
*lp, int (* ListDataCmp)()) : adds
element in order*/
List LstSrc();
List RLstSrc();
```

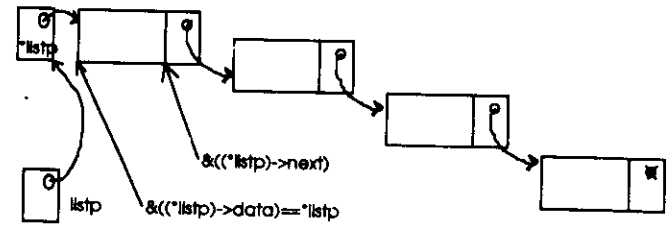
```
/*List LstSrc(List l, ListDataPattern p,
int (*ListDataMatch)()) searches element
matching pattern */
int LstPrn();
int RLstPrn();
/* int LstPrn(List l, int
(*ListDataPrnt)()) prints whole
list l*/

List LstDel();
List RLstDel();
/* List LstDel(List *lp)deletes
whole list  lp */
#endif
```

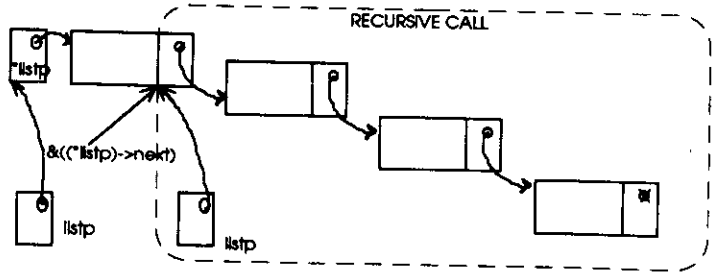
Implementation: list.c

```
#include "mylist.h" /* the typedefs */
#include "list.h"
#define NULL 0
/* adds a new element before an
existing (possibly empty) list;
if successful returns the new list, else
NULL; the input list argument is modified
*/
List AddBfr(data, listp)
ListData *data;
List *listp;
{   List t;
    if(t=(List)malloc(sizeof(ListNode))){
        t->next= *listp;
        *listp=t;
        ListDataCpy(&(*listp)->data, data);
    }
    return t;
}
```

```
/* Returns a pointer to the tail of a
list (all elements except
the first one . On empty list, returns
empty list */
#define Tailp(x) (!(x)?(x):&((x)->next))
/* Remove the first element of a list.
Return pointer to tail */
List RmFrst(listp)
List *listp;
{   List t;
    if(!listp||!*listp)return NULL;
    t=(*listp)->next;
    free(*listp);
    *listp=t;
    return t;
}
/*add one element with a given data field
to a list. The list is supposed to be
sorted according to the comparison
function ListDataCmp (returning -1 for
less, 0 for ==, 1 for >), and the new
element is inserted preserving sorting */
List
AddSrt(data,listp,ListDataCmp)
ListData *data;
List *listp;
int (*ListDataCmp)();
{   int t=1;
    if(!listp)return NULL;
    while((*listp)!=NULL &&
          (t=(*ListDataCmp)(
             &((*listp)->data),data
          ))<1
    )listp= &((*listp)->next);
    if(t)return AddBfr(data,listp);
    else return *listp;
}
```




```
/*Recursive implementation of the same */
List RAddSrt(data,listp,ListDataCmp)
ListData *data;
List *listp;
int (*ListDataCmp)();
{   int t;
    if(!listp)return NULL;
    if(*listp==NULL ||
       (t=(*ListDataCmp)
             (&((*listp)->data),data)
       )>0)
       return AddBfr(data,listp);
    else if(t==0)return *list /*why can I
          be sure t is defined?*/;
    else RAddSrt(data, &((*listp)->next),
                       ListDataCmp);
}
```

```
/* searches a list, returning a pointer
to the first element whose data field
matches a pattern.
Pattern is of type ListDataPattern;
matching is defined by a function
ListDataMatch(datap, patternp)
Listdata *datap;
ListDataPattern *patternp;
returning 1 if they match, 0 otherwise;
    Example: if Listdata is
    char Listdata[100],
    could be char ListDataPattern[10], and
    ListDataMatch could be
    {return strncmp(*datap, *patternp,
    9)==0}
*/
List LstSrc(pattern, list, ListDataMatch)
List list;
ListDataPattern *pattern;
int (*ListDataMatch)();
{
    while(list &&
          !(*ListDataMatch)(
             &(list->data),pattern
          )
    ) list= list->next;
    return list;
}
```



```
/*recursive version of the same */
List RLstSrc(pattern, list,
ListDataMatch)
List list;
ListDataPattern *pattern;
int (*ListDataMatch)();
{   if (list==NULL)return NULL;
    if((*ListDataMatch)(
              &(list->data), pattern
          )
    ) return list;
    return RLstSrc(
       pattern,list->next,ListDataMatch);
}
/*print all elements of list, according
to printing function ListDataPrnt*/
#include <stdio.h>
LstPrn(list,ListDataPrnt)
List list;
int (*ListDataPrnt)();
{   while(list){
    if(ListDataPrnt(list->data)==EOF)
       return EOF;
    list=list->next;
    }
    return 1;
}
/*Recursive version of the above */
RLstPrn(list,ListDataPrnt)
List list;
int (*ListDataPrnt)();
{       if(!list)return 1;
        if(ListDataPrnt(list->data)==EOF)
        return EOF;
   RLstPrn(list->next,ListDataPrnt);
}
```

```
/* Delete a list */
List LstDel(listp)
List *listp;
{       if(!listp)return NULL;
   while (*listp) RmFrst(listp);
   return NULL;
}
/* Recursive version of the above */
List RLstDel(listp)
List *listp;
{       List RRLstDel();
   if (!listp||!*listp)return NULL;
   RRLstDel(listp);
}
static List RRLstDel(listp)
List *listp;
{       if((*listp)->next==NULL){
      free(*listp);
      *listp=NULL;
   }else RRLstDel(&((*listp)->next));
}
```

Customization file:

mylist.h

```
#ifndef MYLIST_H
#define MYLIST_H
typedef char ListData[100];
typedef  char ListDataPattern[10];
#endif
```

Program file

mainlist.c

```
#include <stdio.h>
#include <string.h>
#include "mylist.h"
#include "list.h"
/* this function needed by AddBfr*/
int ListDataCpy(a,b)
```



```
ListData *a,*b;
{strncpy(*a,*b,100);
}
/* These functions passed as arguments to
list routines */
int listdataprint(p)
List p;
{       return printf("%s\n",p->data);
}
int stringmatch(a, b)
char *a, *b;
{ return strncmp(a,b,4)==0;
}
/* main: tests list routines . Input is
assumed to contain the names of the members
of Scientific Computing at ICTP*/
main()
{     List list,p;
   char buf[100];
   int c;
/*create list */
   while ((c=scanf("%s",buf))==1)
      AddSrt(buf,&list,strcmp);
/*display */
   printf("Test displaying: \n");
   printf("Non-recursive version\n");
   LstPrn(list, listdataprint);
   printf("Recursive \n");
   RLstPrn(list,listdataprint);
   printf("NON RECURSIVE SEARCH\n");
/*search for too small than */
   if(LstSrc("aaa",list,stringmatch)!=NULL)
      fprintf(stderr,
      "found non-existent low value\n");
   else
      fprintf(stderr," low-value ok\n");
/*search for too big */
```

```
    if (LstSrc("zzz",list,stringmatch)!=NULL)
       fprintf(stderr,
       "found non-existent high value\n");
    else
       fprintf(stderr," high-value ok\n") ;
/*search for non-existent in-between */
    if (LstSrc("fff",list,stringmatch)!=NULL)
       fprintf(stderr,
    " found non-existent mid value\n");
    else
       fprintf(stderr," mid value ok\n");

/*search for first*/
    if(
    (p=LstSrc("alvi",list, stringmatch))==NULL
    )
       fprintf(stderr," first not found\n");
    else
       listdataprint(p);
/*search for last*/
    if(
    (p=LstSrc("zorz",list,stringmatch))==NULL
    )
       fprintf(stderr," last not found\n");
    else
       listdataprint(p);

/*search for mid*/
    if(
    (p=LstSrc("marc",list,stringmatch))==NULL
    )
       fprintf(stderr," mid not found\n");
    else
       listdataprint(p);


  printf("RECURSIVE SEARCH\n");
```

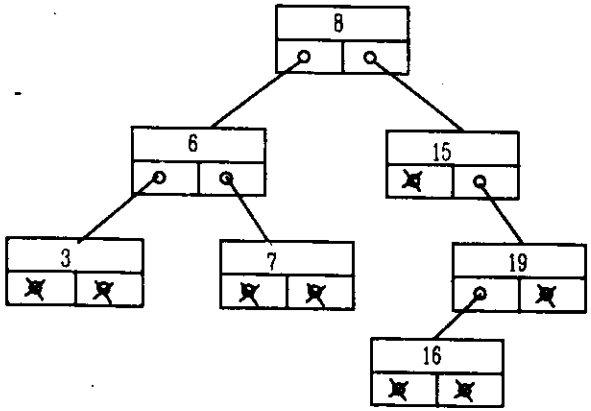


```
/*search for too small than */
    if (RLstSrc("aaa",list,stringmatch)!=NULL)
       fprintf(stderr,
       " found non-existent low value\n");
    else
       fprintf(stderr," low-value ok\n");
/*search for too big */
    if
(RLstSrc("zzzz",list,stringmatch)!=NULL)
       fprintf(stderr,
       " found non-existent high value\n");
    else
       fprintf(stderr," high-value ok\n");
/*search for non-existent in-between */
    if (RLstSrc("fff",list,stringmatch)!=NULL)
       fprintf(stderr,
       " found non-existent mid value\n");
    else
       fprintf(stderr," mid value ok\n");

/*search for first*/
    if (
    (p=RLstSrc("alvi",list,stringmatch))==NULL
    )
       fprintf(stderr," first not found\n");
    else
       listdataprint(p);
/*search for last*/
    if (
    (p=RLstSrc("zorz",list,stringmatch))==NULL
    )
       fprintf(stderr," last not found\n");
    else
       listdataprint(p);

/*search for mid*/
    if (
```

```
    (p=RLstSrc("marc",list,stringmatch))==NULL
    )
       fprintf(stderr," mid not found\n");
    else
       listdataprint(p);
/*  removal of first*/
    p=RmFrst(&list);
    printf(" after removal of first :\n");
    RLstPrn(list,listdataprint);
    if(p!=list)printf(stderr,
       "returned list != changed input\n");
/*  list delete*/
    LstDel(&list);
    if(list!=NULL)
       fprintf(stderr,"Lstdel failed\n");
}
```

Recursive and non-recursive implementations: is recursion essential?

Binary trees




```
structure tree_node{
    int data;
    struct tree_node *left,*right;
}

typedef struct tree_node *Tree;

/* print a binary tree of the above
type in ascending order */
tree_print(tree)
Tree tree;
{   if(tree){
        tree_print(tree->left);
        printf("%d\n",tree->data);
        tree_print(tree->right);
    }
}
```

Simple!

```
/*return a pointer to a tree node
whose data field equals the one passed
as argument */
treeSrc(data,tree)
int data;
Tree tree;
{   if(!tree)return NULL;
    if(tree->data==data)return tree;
    if (tree->data>data)
        return treeSrc(data,tree->left);
    else
        return treeSrc(data,tree->right);
}
```

```
/* add entry to a tree */
Tree tree_add(tree,new)
Tree *tree;
int new;
{
    if(!tree){ return NULL}
    if(!*tree){
        if(
         *tree=(Tree*)malloc(sizeof(tree_node))
        ){/*create new node, attaching
            it to the current "tree" */
            (*tree)->data=new;
            (*tree)-left=(*tree)->right=NULL;
        }
        return *tree;
    }
    if((*tree)->data>new)
        return tree_add(&(*tree)->left,new);
    else if ((*tree)->data<new)
        return tree_add(&(*tree)->right,new);
    else return *tree;
    /*ignore multiple entries*/
}
```

Comment:

- **Recursion sometimes difficult to figure out, but then tends to give simple programs**
- **For trees and other data structures, only possible way of programming**
- **Can be less efficient than loops , when loops possible**

Final comment on dynamic data structures:
PLENTY:

- **Doubly linked lists**
- **Queues (FIFO)**
- **Stacks(LIFO) etc.**

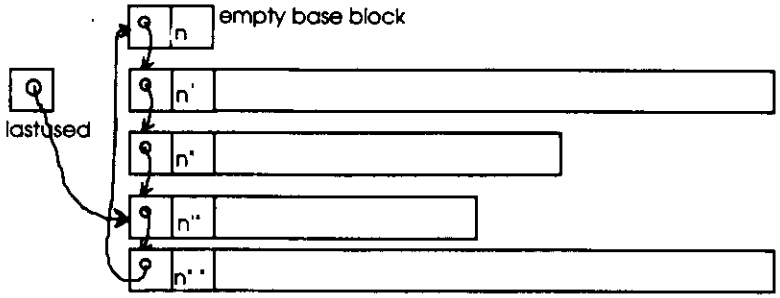
Final example: a realistic memory allocator
Assumption:

> **O.S. can provide a block of memory, but this is an expensive operation**

Implement efficient memory allocator:

- `malloc(n)` **: returns a block of n bytes taking from a list of free blocks or from O.S. if needed**
- `free(p)`**: (re)attaches the free block pointed to by** `p` **to the list of free blocks**

Let O.S. interface be hidden in a function ("morememory") that attaches new block directly to free list.



Memory structure:

- **list of free blocks;**
- **each block composed of a header containing pointer to next and size of block, suitably aligned, followed by free space;**
- **each block's size is a multiple of the header's and the block can be treated as an array of headers;**
- **List is circular; a moving pointer ("lastused") remembers the last block allocated, and start next search from it.**

OPERATION

- **Initially free list is empty;**
- **created in the same way as it is expanded later, when needed.**

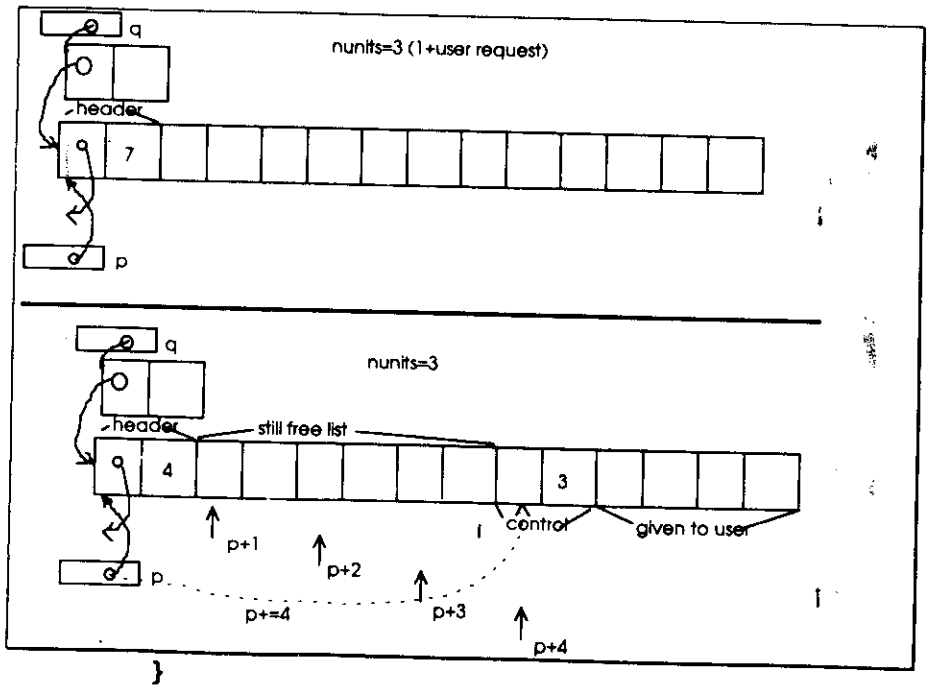
/*Data structures*/

```
typedef ALIGN int; /* assume your
machine wants int and double aligned
to word : MACHINE DEPENDENCY*/
union header{
    struct { union header * next;
             unsigned size;
           } s;
    ALIGN align;
}/* union forces both components to be
aligned to a memory boundary good for
both, that is to a word because of
ALIGN (that is int)hiding machine
dependency*/
typedef union header HEADER;

static HEADER base;/*empty list to
start*/
static HEADER *lastused=NULL; /* last
allocated block*/
```

/*Programs*/

```
char *alloc(nbytes)
unsigned nbytes;
{   HEADER *morecore();
    register HEADER *p, *q;
    register int nunits;
    nunits=1+ (nbytes+sizeof(HEADER)-1) /
        sizeof(HEADER )/*size of block to be
        found, in units of headers */;
    if((q=lastused)==NULL){/*first call*/
        base.s.next=lastused=q=&base;
        base.s.size=0;
        /* now go to search, fail and goto
        list expand*/
    }
```

```
    /*search block of suitable size*/
    for(p=q->s.next;;q=p,p=p->s.next){
    /* follow list with p one forward
       respect to q, starting from lastused
       ending when fail
    */
       if(p->s.size>=nunits){/*found*/
          if(p->s.size==nunits){/*exactly*/
             q->s.next=p->s.next;
             /*unlink *p */
          }else{ /* give to user tail of
                 block found */
                 p->s.size -= nunits;
                 /*block at p has been cut*/
                 p += p->s.size;
                 p->s.size=nunits;
                 /* treating block at p as
an array of cells of type HEADER */
```




```
    }
```



```
          lastused=q;
          return (char*) (p+1);
       }
       /*if here, no suitable block found*/
       if(p==lastused) /*wrap-around, no blocks
                       of that size, ask O.S. */
          if((p=morememory(nunits))==NULL)
             /* if here, nothing to do, else
                p points to right block */
             return NULL;
       }
    }
```

```
/*morememory returns a chunk of memory,
rounded to a suitable value NALLOC, using
O.S. call sbrk. sbrk returns a pointer or
the int -1 if it fails */
#define NALLOC 128
static HEADER * morememory(nu)
unsigned nu;
{
    char *sbrk();
    register char *cp;
    register HEADER *up;
    register int realnu;
    realnu=NALLOC*((nu+NALLOC-1)/NALLOC)
    cp=sbrk(realnu*sizeof(HEADER));
    if((int)cp==-1)  return NULL;
    up=(HEADER*)cp;
    /*create free list header*/
    ip->s.size=realnu;
    /*attach to free list like any other
    block: careful, free assumes header
    is 1 position before the pointer it
    receives, that is user-level pointer*/
    free((char*)(up+1))/*explicit casting to
       protect from machine dependencies*/;
    return(lastused);/*lastused updated by
free*/
}
```



```
free(userp)
/*not optimal but simple; for better  one, see Kernighan and
Ritchie*/
char *userp;
{
    register HEADER *p,*q, *q1;
    p=(HEADER*)userp-1;
    /*points to block header*/
    /* now try to merge this block with
    some other block in free list */
    m=2;/*at most 2 merging can happen */
    q=lastused; q1=q->s.next
    while(1){
    if(q1+q1->s.size==p){
       q1->s.size+=p->s.size;/*merge*/
       p=q1;/* p points to merged block*/
       q->s.next=q1->s.next;/*unlink*/
       q1=q1->s.next;
       m--;
    }else if(p+p->s.size==q1){
       p->s.size+=q1->s.size;/*merge*/
       q->s.next=q1->s.next;/*unlink*/
       q1=q1->s.next;
       m--;
    }else if(m==0|| q1==lastused) {
       /*wraparound or no more merges
       possible: just link*/
       q->s.next=p;
       p->s.next=q1;
       p=(HEADER*)0;
       break;
    }else{
       q=q1;
       q1=q1->s.next;
    }
lastused=q;
}
```