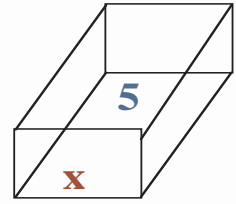


Imperative PLs

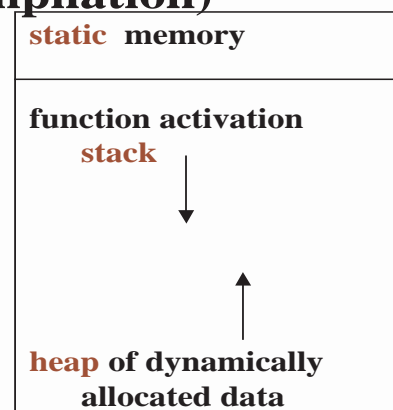
- **Assignment as main operation** $x=w;$
 - Names *refer to Locations containing Values*
 - **L-value:** address of memory location
 - **R-value:** value of an expression



(L-values can be expressions too: $*(f(x)+1)[j] = 5;$)

C program

- **top level type, variable and function declarations**
`typedef int hours; int x; hours tic_toc(void){...}`
- **nested type and variable (but not function) declarations**
- **macros (expanded before compilation)**
- **storage management:**



C Datatypes

- **Primitive: char, int, double, float (+others)**
 - any nonzero value is *true*; zero is *false*
`#define BOOL int; #define TRUE 1; #define FALSE 0;`
- **Aggregates**
 - arrays -- homogeneous (typeof value needs to be known at compile time)
`char a[10]; int b[2][10];` but also `int c[][10]` tfor parameters passed in;
accessed as `b[i][j]`
 - Structures: heterogeneous aggregates

```

struct employee{
    int age;
    double payrate;
} joe
accessed as joe.age

struct rectangle{
    struct point p1;
    struct point p2;
} rct1

```

NB: name of this type is **struct employee**
Not employee

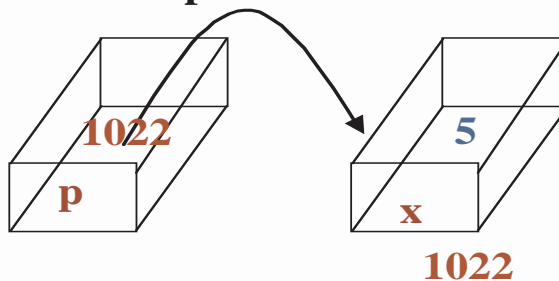
C Datatypes

- **Pointers**: variables whose value is the L-value of a variable
 - declaration: `int *p;`
 - address-of operator `&`: return L-value of variable
 - dereference operator for a pointer: `*` obtains its R-value

```

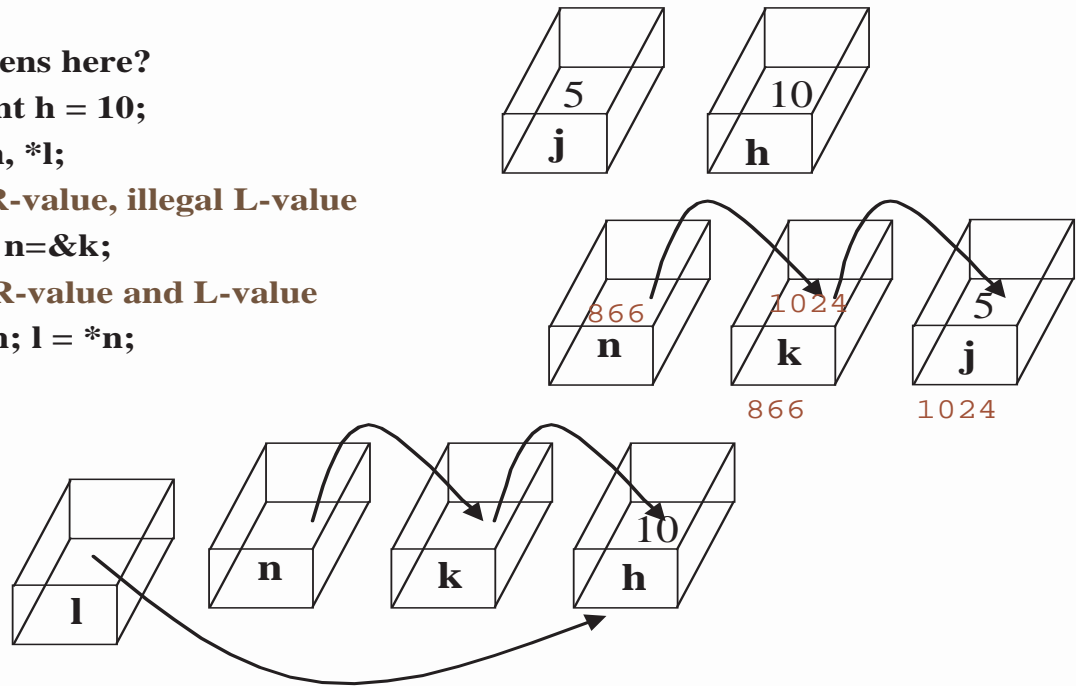
int x = 5;
int *p;
p = &x;
*p = 5;

```



Pointer Expressions

- What happens here?
`int j = 5; int h = 10;`
`int *k, **n, *l;`
`&j, legal R-value, illegal L-value`
`k = &j; n=&k;`
`*n, legal R-value and L-value`
`*n = &h; l = *n;`

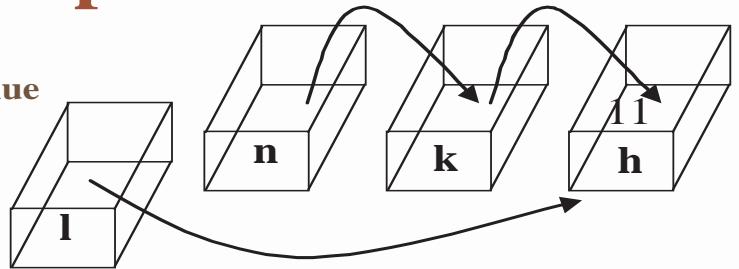


C1, Ryder/Borgida

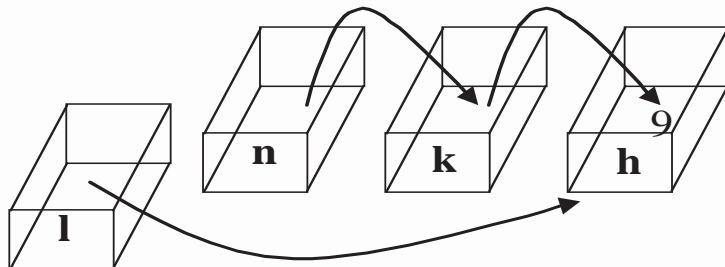
5

Pointer Expressions

- `**n, legal R-value, illegal L-value`
`h = **n+1;`



- `*k, legal R-value, illegal L-value`
`h = *k - 2;`



C1, Ryder/Borgida

3
6

Multipleptr.c

```
#include<stdio.h>
/*program to show multiple levels of dereference*/
main( )
{
    int j, k, l;
    int *q;
    int **s;
    j = 99;
    q = &j; /* q = j is ILLEGAL */
    s = &q; /* s = q is ILLEGAL */
    /*all these names **s, *q, j are synonyms or aliases
    for the same storage location at this program point*/
    printf(" %d %d %d\n",**s, *q, j);
}
/* output
!c> gcc multipleptr.c
!c> a.out
99 99 99 */
```

Space management

- **sizeof(<type>);**
 - sizeof(char)~1 sizeof(int)~4 ?8??
 - sizeof(T *)~ 4
 - sizeof(int [20]) ~ 20 x sizeof(int)
- **sizeof <expression>;**
given *char *s;*
sizeof s ~ 4 sizeof *s ~1
- **Allocating space: malloc**
given *T *p;*
use p = (T *) malloc(sizeof(T))
- **Freeing up space: free free(p);**

Structures and pointers in C

- For structure

```
typedef struct {
    int age;
    double payrate;
} Employee;
```

```
Employee joe, * margePt;
```

- allocate space: not needed for **joe**

```
margePt =
    (Employee*)malloc(sizeof(Employee))
```

- access fields:

```
joe.age
```

```
(*emps).age or emps -> age
```

Pointer Arithmetic

Given declaration **T *pt**; the expression **pt + k** points to location **pt + k × size(T)**

```
int j ;
int *k;
k=&j; /* k is legal L-value */
*k+2 means ((*k)+2) == 5+2 == 7
== (*k+2)
```

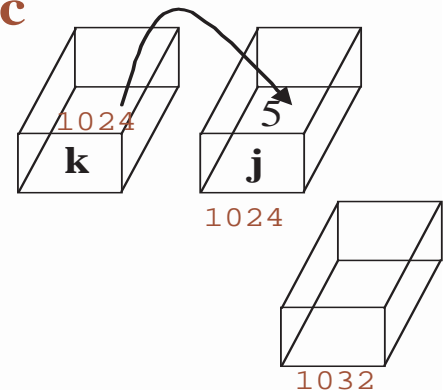
```
//legal R-value (h = *k+2;) // illegal L-value
```

k+2 == 1024 + 2*sizeof j == 1024+2*4 = 1032 has type (**int***), so it is a *legal* (but not meaningful) L-value

*/*need to know layout of storage to see to what (k+2) points: the byte that is 8 bytes beyond the L-value of k, since adding 2 is like adding storage for 2 int (4 bytes each)*/*

```
*(k+2) == *(1032) == ?? /* illegal L-value */
```

k++ has same properties as k



newpointerarith.c

```
struct person{
    int age;
    int socsecnum;
    int phoneno;
};

main( )
{   int a[5], j;
    int *pa, *pb;
    struct person people[3];
    struct person *zz;

    for (j = 0; j < 6; j++)
        a[j] = j; /* initialize a */
```

a, array of ints;
people, array of struct persons

newpointerarith.c

```
int j=0;
while (!feof(stdin) && j<3)
{scanf("%d%d%d", &(people[j].age),
&(people[j].socsec), &(people[j].phoneno));
    printf("output with array elements %d %d %d\n",
        (people[j].age, (people[j]).socsecnum,
        people[j]).phoneno);

    j++;
} /* can I use people[j]->age? Why or why not? */
/* output:
52 999 3699 26 111 5430 24 222 3361 --I typed this at the
output with array elements 52 999 3699 terminal
output with array elements 26 111 5430
output with array elements 24 222 3361
*/
```

```
struct person{
    int age;
    int socsec;
    int phoneno;
} people[3], *zz;
```

newpointerarith.c

```
/* wow. this works! */
printf("\n");
zz = people; /*remember people is an array*/
for (j = 0; j<3; j++)
    { printf("output with pointer %d %d %d\n",
            (*zz).age, zz->socsec, zz->phoneno);
      zz = zz + 1;
    }
printf("\n");
/* output:
output with pointer 52 999 3699
output with pointer 26 111 5430
output with pointer 24 222 3361 */
```

list.c, cont.

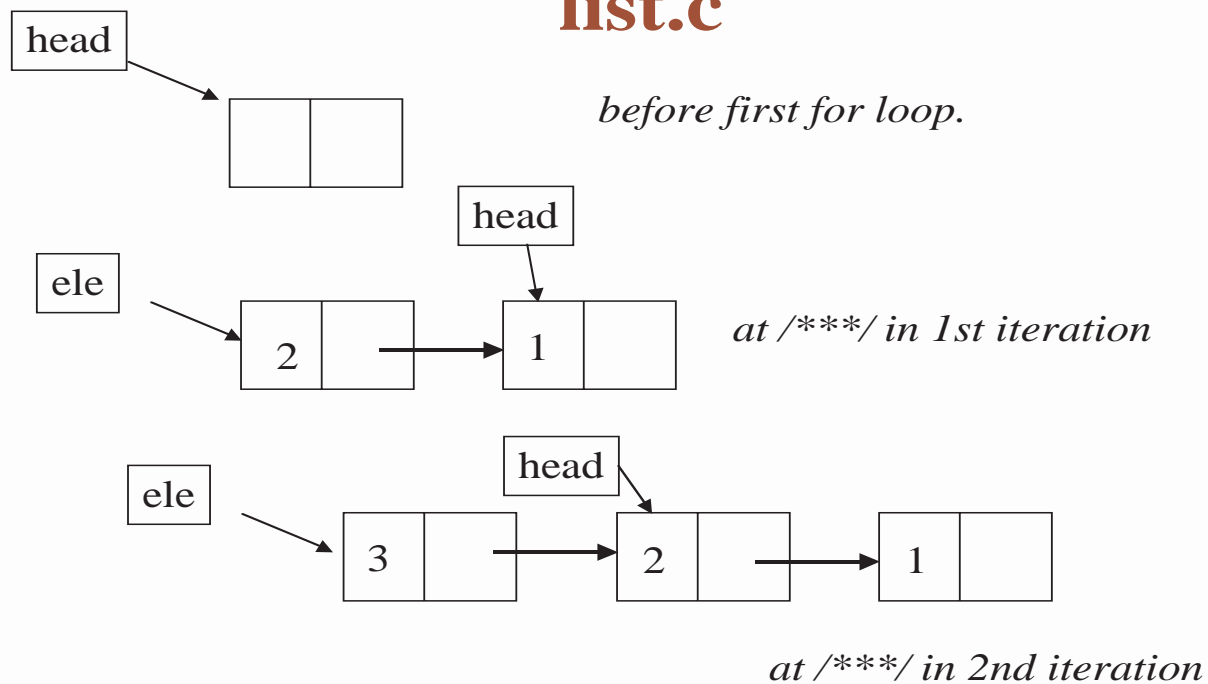
```
main(void)
{ int j;
/*create first node in list*/
  head = (listcell *) malloc(sizeof (listcell));
  head->next = NULL;
/*now create other entries in list of numbers from 1
to 10*/
  head->num = 1;
  for (j=2; j<11; j++)
  {   ele = (listcell*) malloc(sizeof (listcell));
      ele->num = j;
      ele->next =head; /***/
      head = ele;
  }
/*now traverse the list and print the elements*/
  for (p=head; p!=NULL; p=p->next)
      printf("%d ", p->num);
  printf("\n");
}
```

list.c

```
/*sample C program to construct a linear linked list
  of integers built as in Java, adding new elements on
  the front*/
#include<stdio.h>
/*this makes these definitions and variables globals*/
typedef struct cell
    {int num;
      cell *next;
    } listcell ;
listcell *head, *ele, *p;
```

```
typedef struct
    {int num;
      cell next;
    } cell
Why is this illegal?
```

list.c



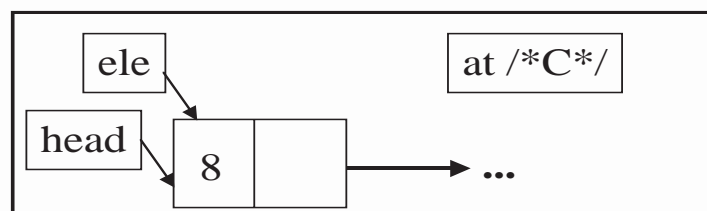
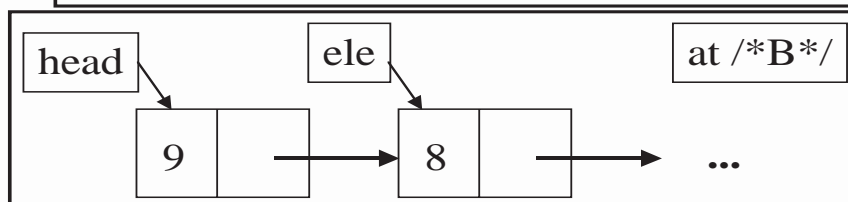
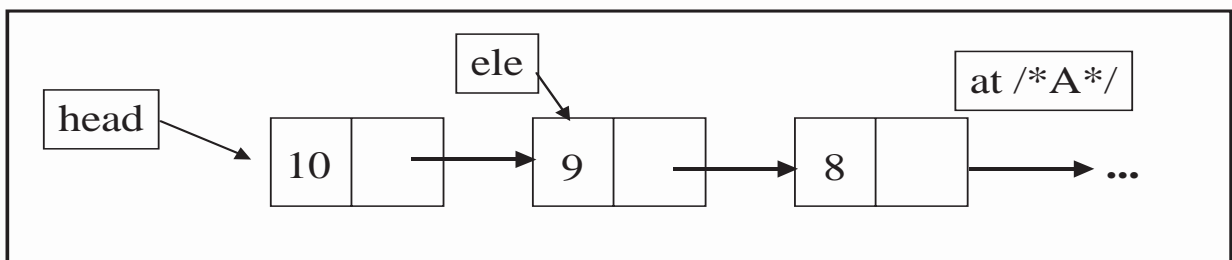
listwithfree.c - deallocating store

list.c followed by new part:

```
/*now delete the first 2 elements of the list and
free their storage */
ele = head->next; /*A*/
free (head); /* free 1st list element storage*/
head = ele;
ele = head->next; /*B*/
free (head); /* free 2nd list element storage*/
head = ele; /*C*/

/*now traverse the list and print the elements*/
for (p=head; p!=NULL; p=p->next)
    printf("%d ",p->num);
printf("\n");
}
```

Trace



listwithfree.c, cont.

```
/* output
!c> a.out
10 9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
!c> */
```

list.c output

```
!c> gcc list.c
!c> ./a.out
10 9 8 7 6 5 4 3 2 1
!c>
```

Pointers and Arrays

- An array name is considered a pointer to its first element
- Suppose you are given the declaration `int a[20];`
 - `a` is pointer to `a[0]` (so type of `a` is `int*`)
 - `pa = &a[0]` and `pa=a` mean the same thing (so type of `pa` is `int*`)
 - `a+1` means L-value of `a[0]` plus as many bytes as are needed to store a value of one of the elements of `a`'s type
 - Pointer arithmetic is an address calculation with respect to the underlying architecture
 - `pb = &a[2];` and `pb= pa+2;` are equivalent (and type of `pb` is `int*`)
 - you can then even do `pb[3]` (`== a[2+3]=a[5]`)
- An array name is not a variable though :-(
 - `a++` and `a = pa` are ILLEGAL!

Array/pointer processing

```
for(a=0; a<10; a++){  
    array[a] = 0;  
}
```

Requires computing `array+a*sizeof array` ; this cannot be done at compile time (a is not known) so it requires run-time multiplication;

```
int *ap;  
for(ap=array; ap < array + 10; ap++){  
    *ap = 0  
}
```

The `ap++` only requires computing `+ 1*sizeof array` , which is known at compile time. (`*ap` is just one dereference).

So the later saves on integer multiplication every time through the loop.

Arrays & pointer examples

```
int array[10];
int *ap; *ap = array +2;
```

Express each of the following expressions in terms of array[]

ap // = &array[2]

***ap** // = array[2]

ap[0] // = (*array + (0)) = *ap+0 = array[2]

ap + 6 // = (array + 2+6) = &array[8] pointer arithmetic

***ap + 6** // = (*ap) + 6 = array[2] + 6 // an integer

***(ap + 6)** // = *(ap+6) = array[8] //an int

&ap // =?? No expression in terms of array

ap[-1] // = array[1]

ap[6] // = array[8]

ap[8] // = array[10] !past end of declarations so it points who-knows where
!!!

Pointers VERSUS References

- **Need explicit dereference operator ***
- **Can mutate R-value of a pointer through pointer arithmetic**
p = p +3;
- **Casting means type conversion of kind of value pointed to **x=(int) (&w)****
- **Special relation to arrays**
- **Intimately tied to dynamic storage allocation**
- **Are implicitly dereferenced**
- **Cannot mutate R-value of a reference**
- **Casting just satisfies the type checker; does no type conversion**
- **No special relation to arrays**

Heap Storage

- Allocation

```
void * malloc ( <#of bytes> )
```

- returns pointer to block of contiguous storage of n bytes (chars), if available;
- if not enough memory left for allocation, *malloc* returns NULL pointer
 - So you ALWAYS have to check return the value
- to allocate storage for some type T requires giving *malloc* the proper amount of bytes needed, and casting the pointer returned

```
head = (listcell *) malloc(sizeof (listcell));
```

or

```
head = (listcell *) malloc(sizeof head );
```

- Deallocation

```
void free( <starting Address> )
```

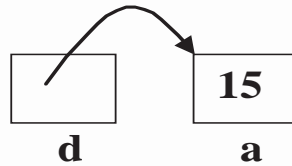
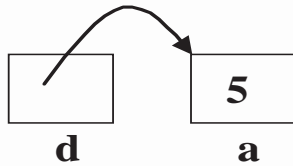
- heap manager keeps track of space allocation requests, to know amount to free

Stack vs Heap

- *Procedure activations, statically allocated local variables, parameter values*
- *Lifetime same as block in which variables are declared*
- *Stack frame with each invocation of procedure*
- **Dynamically allocated data structures, whose size may not be known in advance**
- **Lifetime extends beyond block in which they are created**
- **Must be explicitly freed or garbage collected**

Problems with Pointers

- **Null** doesn't point to anything by definition so cannot dereference it
 - `a = 0;` /* makes `a`'s value NULL */
 - `if (a == NULL) ...` /* tests for a NULL pointer value*/
- **Multiple level pointers and pointer arithmetic** can lead to trying to access locations outside
 - Can be used as L-values or R-values



```
program
int a;
int *d;
d = &a;
a = 5;
*d = 10 + *d;
a = 10 + 5
```

C1, Ryder/Borgida

27

More Problems with Pointers

- **Dangling pointer**
 - Storage pointed to is freed, but pointer is not set to NULL
 - Then you are able to access storage whose values may no longer be the same (re-allocated)
- **Garbage**
 - Pointer itself is freed (perhaps by execution going out of its declaring scope) but heap locations pointed to are not freed. Then, there is no way to access this heap storage: **memory leak**

C1, Ryder/Borgida

4
28